OULUN AMMATTIKORKEAKOULU

Sami Varanka

KOOSTEOPINNÄYTETYÖ: C++:N MUISTINHALLINTA JA LIITÄNNÄISTEN TOTEUTTAMINEN SEKÄ MOBIILIPELIN KEHITYS UNITY-PELIMOOTTORILLA

KOOSTEOPINNÄYTETYÖ: C++:N MUISTINHALLINTA JA LIITÄNNÄISTEN TOTEUTTAMINEN SEKÄ MOBIILIPELIN KEHITYS UNITY-PELIMOOTTORILLA

Sami Varanka
Opinnäytetyö
Kevät 2017
Tietotekniikan koulutusohjelma
Oulun ammattikorkeakoulu

TIIVISTELMÄ

Oulun ammattikorkeakoulu
Tietotekniikan koulutusohjelma, ohjelmistokehityksen suuntautumisvaihtoehto

---

Tekijä(t): Sami Varanka
Opinnäytetyön nimi: Koosteopinnäytetyö: C++:n muistinhallinta ja liitännäisten toteuttaminen sekä mobiilipelin kehitys Unity-pelimoottorilla
Työn ohjaaja(t): Veijo Väisänen, Pertti Heikkilä, Jaakko Kaski
Työn valmistumislukukausi ja -vuosi: Kevät 2017
Sivumäärä: 10 + 3 liitettä

Tämä opinnäytetyö on tehty kolmessa viiden opintopisteen osassa, jottei opinnäytetyön tekeminen olisi liian raskasta opintojen lopussa. Ensimmäisessä osaopinnäytetyössä tutustuttiin C++:n muistinhallintaan ja sitä varten kehitettyihin tekniikkoihin.

Opinnäytetyön toisessa osassa tutustuttiin liitännäisten toteuttamiseen C++-ohjelmointikielellä. Työssä laajennettiin laskinsovellusta lisäämällä siihen liitännäinen, joka mahdollistaa yksinkertaiset vektorilaskut. Liitännäinen toteutettiin DLL-moduulina.

Opinnäytetyön kolmannessa osassa toteutettiin mobiilipeli Android-käyttöjärjestelmälle sekä siihen pisteiden tallennus ja lataus pilvestä. Peli toteutettiin Unity-pelimoottorilla ja pilvitietokantayhteys toteutettiin NodeJS:llä ja JavaScript-ohjelmointikielellä. Pelin tilaajana toimi Pertti Heikkilä.

Opinnäytetyön ensimmäisen ja toisen osan aikana C++-ohjelmointikieli tuli tutummaksi. Kolmannen osan aikana Unity-pelimoottori tuli tutuksi ja opin lisää, kuinka vähentää skriptien välisiä viittauksia. Lisäksi opin lisää NodeJS-ympäristöstä. Kolmannessa osassa tehtyä peliä on tarkoitus jatko kehittää harrastusprojektina.

Asiasanat: C++, Unity, Ohjelmointi, DLL

ABSTRACT

Oulu University of Applied Sciences
Information Technology, Software development

Author(s): Sami Varanka
Title of thesis: Compilation thesis: C++ memory management and software extension implementation and mobile game development using Unity game engine
Supervisor(s): Veijo Väisänen, Pertti Heikkilä, Jaakko Kaski
Term and year when the thesis was submitted: Spring 2017
Pages: 10 + 3 appendices

This thesis was done in three parts. The first part consists of C++ memory management and memory management techniques.

The second part consists of software extension implementation in C++. During the second part simple vector calculator software extension was implemented for basic calculator application found in the book “Programming: Principles and Practices using C++, Second edition” by Bjarne Stroustrup. Software extension was implemented as a DLL-module.

The third part consists of game development for Android operating system using Unity game engine and using cloud database for scoreboard. Cloud database connection was implemented with JavaScript programming language in NodeJS framework.

During the first and the second parts C++ programming language became more familiar. During the third part Unity game engine became familiar and I learned how to easily reduce references between scripts. I also learned more about NodeJS framework. The development of the game made in the third part is going to be continued as a hobby project.

Keywords: C++, Unity, Ohjelmointi, DLL

SISÄLLYS

1 JOHDANTO

Tämä opinnäytetyö on tehty kolmessa viiden opintopisteen osassa. Ensimmäinen osa tehtiin toisen vuoden keväällä, toinen osa tehtiin kolmannen vuoden keväällä ja kolmas osa tehtiin neljännen vuoden keväällä. Kun kaikki osat oli saatu valmiiksi, tehtiin koostedokumentti, jossa esiteltiin kaikki kolme osaa ja ne lisättiin liitteenä siihen.

Työn ensimmäisessä osassa tutustuttiin C++:n muistinhallintaan ja muistinhallintatekniikkoihin. Valitsin ensimmäisen osan aiheeksi C++:n muistinhallinnan, koska pidän C++-ohjelmointikielestä ja halusin oppia lisää muistinhallinnasta. Jälkeenpäin ajatellen ensimmäisen osan aiheeksi olisi voinut valita jonkin helpomman aiheen.

Työn toisessa osassa tutustuttiin liitännäisten toteuttamiseen. Myös toisessa osassa kielenä oli C++. Valitsin aiheen, koska olin ajatellut tekeväni kaikki osat C++-kielellä ja työn aloituksen aikoihin olin tutustunut DLL-moduulien ajonaikaiseen linkittämiseen.

Työn kolmannessa osassa toteutettiin mobiilipeli Unity-pelimoottorilla sekä siihen pisteiden tallennus ja lataus pilvestä. Valitsin tämän aiheen, koska olin ennen työn aloittamista vuoden kehittänyt peliä Unity-pelimoottorilla, joten olin hyvin ehtinyt tutustua Unity-pelimoottoriin. Lisäksi olen kiinnostunut pelien tekemisestä, joten tämä aihe sopi hyvin minulle.

2 ENSIMMÄISEN OSAN ESITTELY

Opinnäytetyön ensimmäinen osa (liite 1) tehtiin toisen vuoden keväällä. Työssä tutustuttiin C++:n muistinhallintaan ja sitä varten kehitettyihin tekniikoihin etenkin muistialtaaseen. Jotta työ ei olisi paisunut liikaa, työssä käsiteltiin vain yhden säikeen muistinhallintaa. Työssä esiintyvien esimerkkien toteuttamiseen on käytetty Microsoftin Visual Studio 2013 -kehitysympäristöä.

Työssä käsitellään muistialueita, viittausten laskentaa, resurssien automaattista hallintaa sekä lopuksi muistiallasta. Resurssien automaattista hallintaa käsittelevän pääluvun yhteydessä esitellään RAII (resource acquisition is initialization) sekä älykkäät osoittimet. Muistiallasta käsittelevän pääluvun yhteydessä pyrittiin tekemään koodipätkä, joka hieman selventäisi muistialtaan käytöstä saatavia hyötyjä.

3 TOISEN OSAN ESITTELY

Opinnäytetyön toisessa osassa (liite 2), joka toteutettiin kolmannen vuoden keväällä, tutustuttiin linkitettäviin kirjastoihin ja liitännäisten toteuttamiseen. Työn osana laajennettiin Bjarne Stroustrupin kirjasta ”Programming: Principles and practices using C++, second edition” löytyvää laskin-sovellusta laajentamalla sitä liitännäisellä, joka mahdollistaa yksinkertaiset vektorilaskut. Liitännäinen toteutettiin DLL-moduulina. Työn toteuttamiseen käytettiin Microsoftin Visual Studio 2013 -kehitysympäristöä ja C++-ohjelmointikieltä.

Työssä esitellään staattisesti linkitettävät kirjastot, mutta pääpaino on dynaamisesti linkitettävissä kirjastoissa, koska ne voidaan lisätä ohjelmaan ajon aikana. Linkitettävien kirjastojen jälkeen työssä kerrotaan liitännäisistä ja niiden toteuttamisesta. Lopuksi työssä esitellään laskimen liitännäisen toiminta ja toteutus.

4 KOLMANNEN OSAN ESITTELY

Työn kolmas osa (liite 3) toteutettiin viimeisen vuoden keväällä. Siinä toteutettiin mobiilipeli Android-käyttöjärjestelmälle käyttäen Unity-pelinkehitysympäristöä sekä pisteiden tallennus ja lataus pilvitietokannasta. Pelissä käytettävät skriptit toteutettiin käyttäen C#-ohjelmointikieltä ja tietokantayhteyden toteuttamiseen käytettiin JavaScript-ohjelmointikieltä. Tietokantana käytetään mLabin MongoDB-tietokantaa, joka käyttää Microsoftin Azure-pilvialustaa.

Työssä kerrotaan ensiksi Unitysta ja MongoDB:stä. Unitysta kerrotaan enemmän, koska työssä keskityttiin enemmän pelien kehitykseen Unity-pelimoottorilla. Unitysta kerrotaan ensiksi sen kehityksestä, jonka jälkeen kerrotaan sen toiminnasta. Unitysta ja MongoDB:stä kertovan pääluvun jälkeen kerrotaan pelin toteutuksesta.

5 YHTEENVETO

Tämä opinnäytetyö on tehty kolmessa osassa yhden ison opinnäytetyön sijaan. Opinnäytetyön ensimmäinen ja toinen osa käsittelevät C++-ohjelmointikieltä, mutta kolmas osa on peliohjelmointia Unity-pelimoottorilla ja siinä käytetään C#-ohjelmointikieltä. Ensimmäisen ja toisen osan väliltä löytyy yhtäläisyyksiä, mutta kolmas osa eroaa niistä.

Opinnäytetyön ensimmäisen ja toisen osan ohjelmointiosuudet olivat vain opinnäytetyötä varten keksittyjä, mutta kolmannen osan peliä on tarkoitus jatko kehittää harrastusmielessä. Peli on tarkoitus saada julkaistua Google Play -kaupassa.

Minulle opinnäytetyön tekeminen kolmessa osassa sopi hyvin. Jos koko opinnäytetyön olisi tehnyt viimeisenä työnä, valmistumiseni olisi myöhästynyt hyvin suurella todennäköisyydellä. Toisaalta, koska opinnäytetyön eri osissa ei tarvitse olla sama aihe, se voi sekoittaa opinnäytetyötä. Myöskään viiden opintopisteen kokonaisuudessa ei tule kovin syvällistä perehtymistä asiaan. Itse yritin pitää aiheeni C++:ssa ja onnistuinkin siinä ensimmäisen ja toisen osan kohdalla. Kolmannen osan kohdalla olin vasta suorittanut valinnaiset Oulu Game Lab -kurssit, joissa kehitettiin peliä Unity-pelimoottorilla.

Sami Varanka

C++:N MUISTINHALLINTA JA MUISTINHALLINTATEKNIIKAT

C++:N MUISTINHALLINTA JA MUISTINHALLINTATEKNIIKAT

Sami Varanka
Opinnäytetyö, ensimmäinen osa
Kevät 2015
Tietotekniikan koulutusohjelma
Oulun ammattikorkeakoulu

SISÄLLYS

1 JOHDANTO

Tietokoneohjelmat käyttävät tietokoneen muistia ja muita resursseja. Kun kaikki ohjelmassa olevat viittaukset näihin resursseihin häviävät ne täytyy vapauttaa takaisin muiden ohjelmien käyttöön. Ylemmän tason kielissä kuten C#, joka toimii Microsoftin .NET ympäristössä, on käytössä automaattinen roskienkeruu, joka huolehtii käyttämättömän muistin vapauttamisesta.

Tässä opinnäytetyön ensimmäisessä osassa tutkitaan C++:n muistinhallintaa ja sitä varten kehitettyjä tekniikoita, etenkin muistiallastekniikkaa, joka on tarkemmin sanottuna muistinvaraustekniikka. Koska nykyajan prosessoreissa on yleensä useampi kuin yksi säie, muistihallintaan kuuluu yhden säikeen lisäksi myös monisäikeinen muistinhallinta. Työstä tulisi kuitenkin liian laaja, joten se on rajattu vain yhden säikeen muistinhallintaan. Valitsin tämän aiheen, koska hyvän C++-ohjelmoijan täytyy tuntea C++:n muistinhallinnasta, jotta hän voi kirjoittaa tehokasta koodia.

C++ on suunniteltu olemaan tehokas ohjelmointikieli, joka toimii lähellä käytössä olevaa laitteistoa, eikä siinä siksi ole käytössä automaattista roskienkeruuta huolehtimassa muistinhallinnasta. Ohjelmoijan täytyy siis itse huolehtia muistinhallinnasta.

2 MUISTIALUEET

C++:ssa muisti jaetaan kolmeen eri alueeseen, joilta muuttujien tilanvaraus tapahtuu. Nämä muistialueet ovat staattinen, automaattinen ja dynaaminen.

2.1 Staattinen

Staattisella muistialueella olevat muuttujat säilyvät koko ohjelman suorituksen ajan ja ovat kaikkien saatavissa.

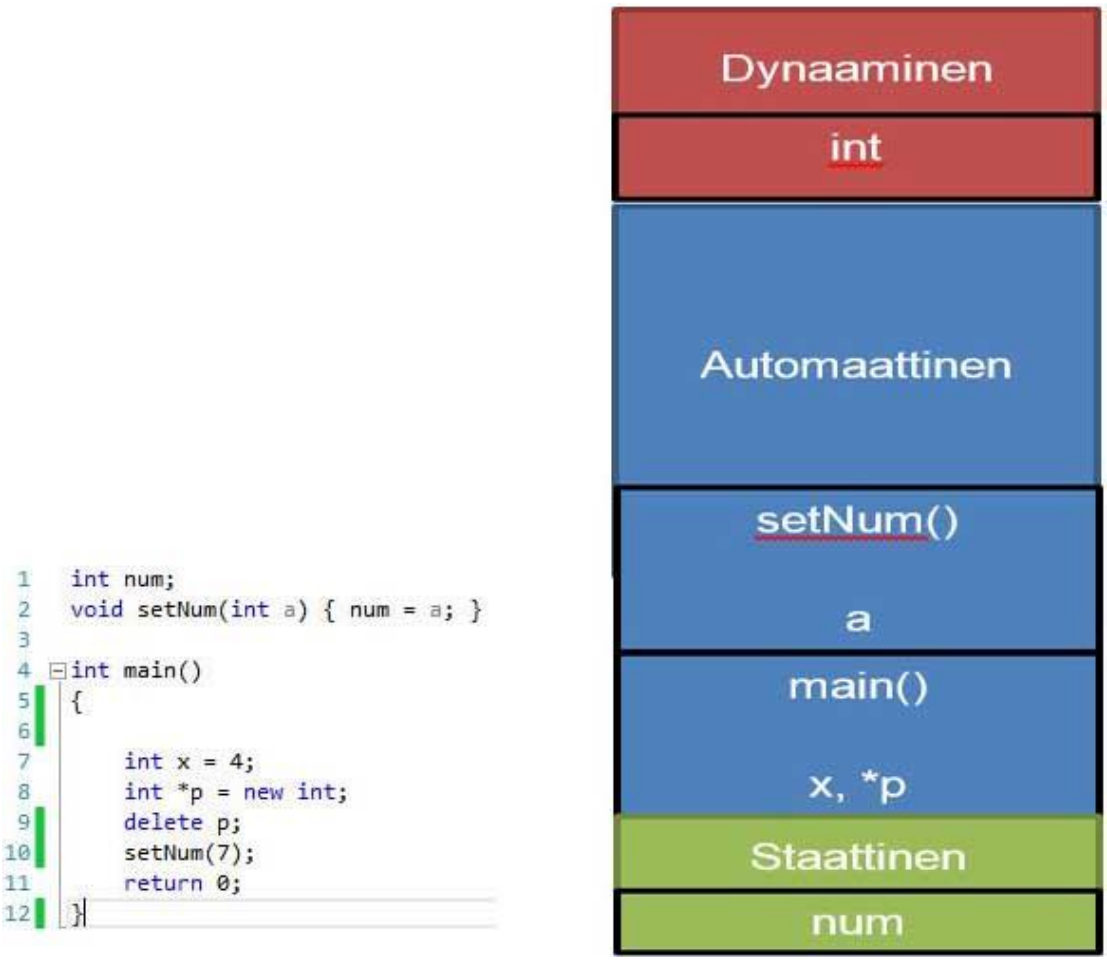
2.2 Automaattinen

Automaattinen muisti on yleensä toteutettu pinomuistina. Tällä muistialueella olevien muuttujien tilanvapautus tapahtuu automaattisesti, kun ohjelmakoodissa siirrytään pois lohkosta, jossa muuttujat on luotu.

2.3 Dynaaminen

Ohjelmoija voi ajon aikana varata muistia dynaamiselta eli vapaalta muistialueelta new-operaattorilla, joka palauttaa osoitteen varatun muistin alkuun. Kun muistia varataan dynaamisesti, se täytyy myös muistaa vapauttaa jossain välissä ohjelman suoritusta delete-operaattorilla, jottei synny muistivuotoja.

Kuvan 1 esimerkistä näkee, kuinka muuttujat ovat asettuneet ohjelman suorituksen aikana eri muistialueille.



KUVA 1: C++:n muistialueet

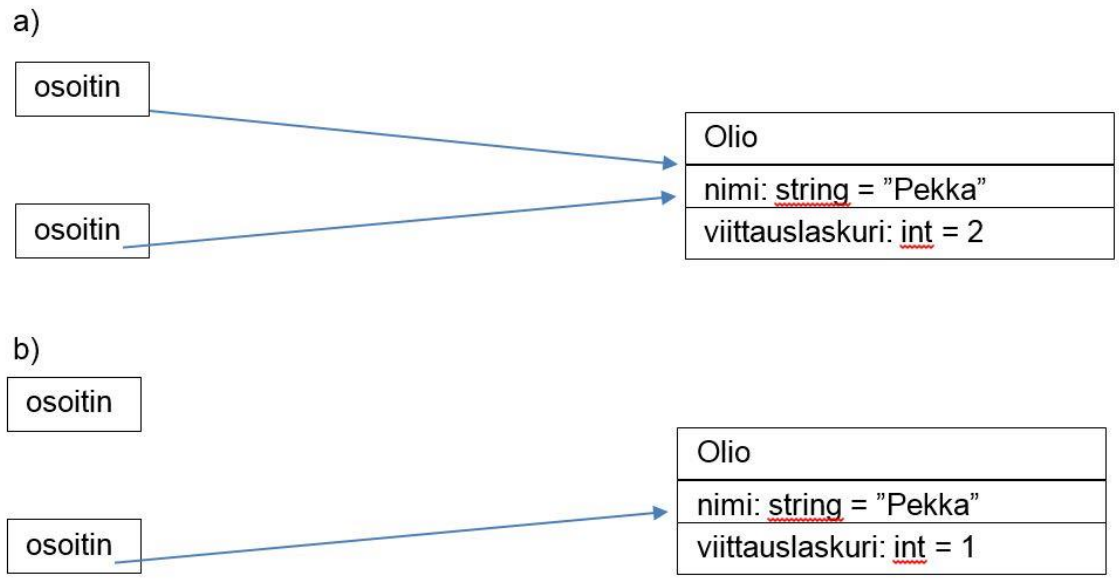
Rivillä 1 luodaan kokonaislukumuuttuja num staattiselle muistialueelle. Seuraavalla rivillä määritellään setNum-funktio, joka asettaa parametrina saadun arvon num-kokonaislukumuuttujaan. Parametrimuuttuja a tuhotaan, kun ohjelman kontrolli poistuu funktiosta. Riviltä 4 alkaa pääohjelman määrittely. Rivillä 7 luodaan kokonaislukumuuttuja x, joka on paikallinen main-funktiossa. Sitten rivillä 8 varataan dynaamiselta muistialueelta tilaa yhden kokonaislukumuuttujan verran ja osoittimeen p sijoitetaan osoite varatun muistilohkon alkuun. Tämän jälkeen rivillä 9 juuri varattu muistilohko vapautetaan. Rivillä 10 kutsutaan setNum-funktiota, jolle annetaan parametrina kokonaisluku seitsemän. Rivillä 11 suoritetaan return-käsky, joka aiheuttaa

poistumisen main-funktiosta ja main-funktiossa määritellyt paikalliset muuttujat tuhotaan. (1.)

3 VIITTAUSTEN LASKENTA

Eräs tapa, jolla voidaan erottaa käyttämätön muisti käytössä olevasta muistista, on käyttää viittausten laskentaa. Viittausten laskennassa jokaisella oliolla on viittauslaskuri, joka kertoo kuinka moni eri osoitin osoittaa siihen.

Aina kun uusi osoitin laitetaan osoittamaan ko. olioon, viittauslaskurin arvoa kasvatetaan yhdellä, kun olioon osoittava osoitin laitetaan osoittamaan jonnekin muualle, viittauslaskurin arvoa vähennetään yhdellä. Kuvasta 2 näkee viittausten laskennan toimintaperiaatteen.



KUVA 2. Viittausten laskennan toiminta

Kuvan a-kohdassa olion viittauslaskurin arvo on kaksi, koska olioon osoittaa kaksi eri osoitinta. Kuvan b-kohdassa viittauslaskurin arvo on yksi, koska toinen osoitin ei enää osoita olioon. Kun viittauslaskurin arvo menee nollaan, ohjelmassa ei ole yhtään viittausta olioon ja olio voidaan tuhota.

4 RESURSSIEN AUTOMAATTINEN HALLINTA

4.1 RAII

Kun ohjelmassa on varattu resursseja, ne täytyy vapauttaa käytön jälkeen, kuten kuvan 3 koodiesimerkissä näkyy.

```
1  void func1()
2  {
3      int *p = new int;
4      int *arr = new int[100];
5
6      //Tee jotain
7
8      delete p;
9      delete[] arr;
10
11 }
12
13 int main()
14 {
15     func1();
16     return 0;
17 }
18
```

KUVA 3. Dynaamisen muistin varaaminen ja vapauttaminen

Esimerkissä func1-funktiossa varataan dynaamisesti muistia yhdelle kokonaisluvulle, jonka jälkeen muistia varataan kokonaislukutaulukolle. Sitten funktiossa tehdään jotain ja lopuksi varattu muisti vapautetaan. Jos funktiossa ”Tee jotain” -kohdassa poistutaan funktiosta, osoittimet varattuihin muistialueisiin menetetään ja syntyy muistivuotoja, koska varatut muistialueet jäävät vapauttamatta. (2.)

Ongelman voi ratkaista sitomalla resurssin käytön pinoon luodun olion elinaikaan. Varatut resurssit vapautetaan olion tuhoajassa. Tätä tekniikka kutsutaan nimellä RAII (Resource Acquisition Is Initialization) Kuvassa 7 on kuvan 6 koodi, mutta nyt resurssit on sidottu olioihin.

```
1  class Olio1
2  {
3  public:
4      Olio1() { p = new int; }
5      ~Olio1() { delete p; }
6  private:
7      int *p;
8  };
9
10 class Olio2
11 {
12     public:
13         Olio2() { arr = new int[100]; }
14         ~Olio2() { delete[] arr; }
15     private:
16         int *arr;
17 };
18
19 void func1()
20 {
21     Olio1 olio1;
22     Olio2 olio2;
23
24     //Tee jotain
25 }
26
27 int main()
28 {
29     func1();
30     return 0;
31 }
32
```

KUVA 4. Resurssien automaattinen vapautus

Esimerkissä on nyt lisätty kaksi luokkaa Olio1 ja Olio2. Olio1-luokasta luotu olio varaa muodostimessaan dynaamisesti muistia yhden kokonaisluvun verran ja vapauttaa varatun muistin tuhoajassaan. Olio2-luokasta luotu olio varaa muodostimessaan muistia 100-paikkaiselle kokonaisluku taulukolle ja vapauttaa varatun muistin tuhoajassaan. Koska molemmat oliot on luotu pinoon, ne tuhotaan ja varatut resurssit vapautetaan, kun ohjelman kontrolli poistuu func1-funktiosta.

4.2 Älykkäät osoittimet

Osoittimia, jotka pystyvät vain osoittamaan muistipaikkaan, kutsutaan paljaiksi osoittimiksi (raw pointer). Paljaiden osoittimien käyttö ei ole suositeltavaa, koska ohjelmoijan täytyy itse huolehtia niiden osoittamien muistialueiden vapauttamisesta ja se altistaa ohjelman muistivuodoille. Paljaiden osoittimien sijasta voidaan käyttää älykkäitä osoittimia (smart pointer).

Älykäs osoitin on luokkapohja (class template), jonka pohjalta luodun luokan olio jäljittelee paljasta osoitinta, niin että sekin osoittaa vain muistipaikkaan, ja sitä voidaan käyttää monella tapaa samalla tavalla. Älykäs osoitin luodaan pinoon ja sille annetaan muodostimeen parametrina paljas osoitin, jonka älykäs osoitin sitten omistaa. Se tarkoittaa sitä, että älykäs osoitin on vastuussa paljaan osoittimen osoittaman muistialueen vapauttamisesta. Älykkään osoittimen tuhoajassa tapahtuu paljaan osoittimen osoittaman muistialueen vapautus delete-operaattorilla. Koska älykäs osoitin on luotu pinoon, sen tuhoajaa kutsutaan automaattisesti, kun ohjelman kontrolli poistuu alueelta, jolla se on määritelty. Näin ollen ohjelmoijan ei tarvitse itse huolehtia muistialueen vapauttamisesta. (1; 2.)

4.2.1 Standardikirjaston yhden omistajan osoitin

Kun halutaan käyttää osoitinta, jolla on vain yksi omistaja, voidaan käyttää std::unique_ptr-luokkapohjaa. Koska osoittimella on vain yksi omistaja, mikään muu luokasta luotu olio ei voi käyttää samaa osoitinta. Osoittimen omistajuus voidaan kuitenkin siirtää toiselle oliolle käyttämällä std::move-funktiota. Kun siirto on suoritettu, alkuperäistä oliota ei voida enää käyttää. Kuvassa 5 näkyy unique_ptr-olion luonti ja sen omistaman osoittimen omistajuuden siirto.

```
std::unique_ptr<Foo> uptr(new Foo);
auto uptr2 = std::move(uptr);
```

KUVA 5. Osoittimen omistajuuden siirto

Ensimmäisellä rivillä luodaan uusi std::unique_ptr-olio. Toisella rivillä osoittimen omistajuus siirretään std::move-funktiolla. (3; 4.)

4.2.2 Standardikirjaston jaettu osoitin

Jos tarvitaan osoitinta, jolla voi olla yhtä aikaa monta omistajaa, voidaan käyttää std::shared_ptr-luokkapohjaa. Luokasta luotu olio voi jakaa omistamansa osoittimen toisen shared_ptr-olion kanssa. Koska osoitinta voi käyttää yhtä aikaa useampi kuin yksi olio, osoittimen osoittaman muistialueen saa vapauttaa vasta viimeinen sitä käyttävä olio. Tietoa siitä, kuinka moni olio käyttää osoitinta, pidetään yllä viittauslaskurin avulla. (3.)

4.2.3 Standardikirjaston heikko osoitin

Joskus olion täytyy päästä käyttämään shared_ptr-olion omistamaa osoitinta ilman, että viittauslaskurin arvoa kasvatetaan. Tällainen tilanne tapahtuu esimerkiksi silloin, kun shared_ptr-olioilla on viittauksia toisiinsa. Silloin voidaan käyttää std::weak_ptr-luokkapohjaa. Aina kun halutaan luoda weak_ptr-olio, se luodaan shared_ptr-oliosta antamalla weak_ptr-luokan muodostimeen parametrina shared_ptr-olio. (4; 5.)

Kuvassa 6 näkyy, kuinka heikko osoitin luodaan. Ensimmäisellä rivillä luodaan std::shared_ptr-olio, ja toisella rivillä luodaan std::weak_ptr-olio.

```
std::shared_ptr<Foo> sptr(new Foo);
std::weak_ptr<Foo> wptr(sptr);
```

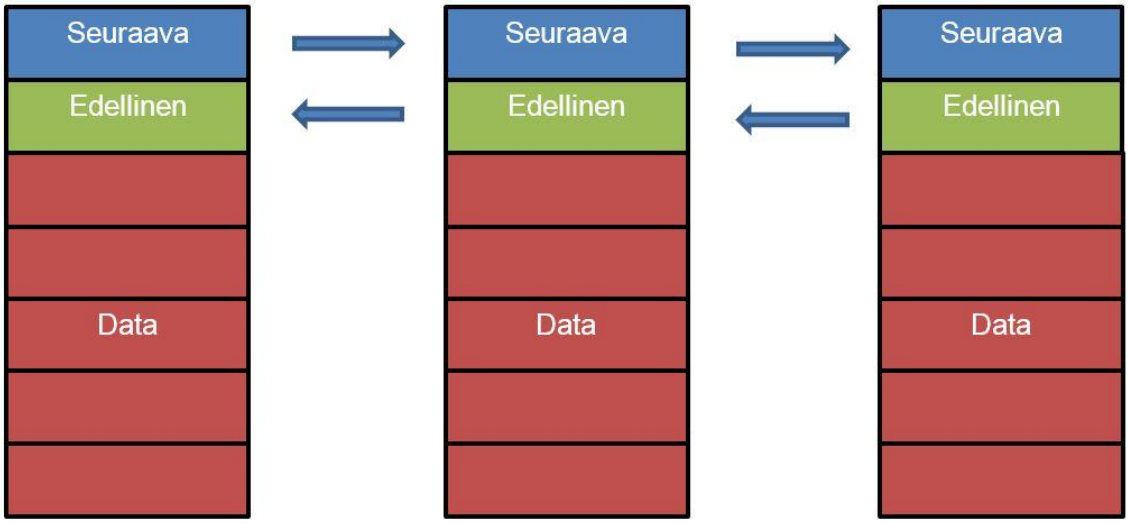
KUVA 6. Heikon osoittimen luonti

5 MUISTIALLAS

Muistin dynaaminen varaaminen voi viedä paljon aikaa. Lisäksi se aiheuttaa muistin pirstoutumista, jolloin ohjelman suoritus hidastuu. Jos ohjelmassa joudutaan tekemään useita dynaamisia muistin varauksia, on suositeltavaa käyttää muistiallasta (memory pool). Muistialtaan käytöstä saataviin hyötyihin kuuluvat muun muassa

- vähäinen muistin pirstoutuminen
- muistin nopeampi varaaminen ja vapauttaminen. (6.)

Kun ohjelma käynnistyy, muistiallas varaa ensiksi yhden ison muistilohkon käyttöjärjestelmältä. Sitten muistilohko jaetaan pienempiin lohkoihin. Lohkoja säilytetään kahteen suuntaan linkitetyssä listassa (doubly linked list), joten jokaisen lohkon alussa täytyy olla osoittimet seuraavaan ja edeltävään lohkoon. Kuva 7 kuvaa kahteen suuntaan linkitettyä listaa.



KUVA 7. Kahteen suuntaan linkitetty lista

Kun ohjelmassa täytyy varata muistia dynaamisesti, se pyydetään muistialtaalta, joka palauttaa osoittimen seuraavaan vapaaseen lohkoon ja merkkaa sen varatuksi. Kun lohko halutaan vapauttaa, se merkataan vapaaksi. Jos listasta ei löydy yhtään vapaata tai sopivan kokoista lohkoa, muistiallas voi varata lisää muistia käyttöjärjestelmältä ja palauttaa osoittimen siihen. (6; 7.)

Kuvan 8 koodiesimerkissä on toteutettu kaksi luokkaa, jotka molemmat käyttävät dynaamisesti varattua muistia. Toisen luokan käyttämä muisti on kuitenkin valmiiksi varattua.

```
1   #include <iostream>
2   #include <ctime>
3
4   char *mlohko = new char[1024];
5   class TestClass1
6   {
7       char x[500];
8   };
9
10  class TestClass2
11  {
12
13  public:
14      void *operator new(unsigned int koko) { return (void*)mlohko; }
15      void operator delete(void *p) {}
16  private:
17      char x[500];
18  };
```

KUVA 8. Muistialtaan periaate

Yllä olevassa kuvassa rivillä 4 on luotu globaali mlohko-niminen osoitinmuuttuja, joka osoittaa char-tyyppiseen muistilohkoon. Se toimii tässä esimerkissä yksinkertaisena muisti altaana. Riveillä 6–8 määritellään luokka, joka käyttää tavallisia new- ja delete-operaattoreita. Riveillä 11–18 Määritellään luokka, joka käyttää ylikirjoitettuja new- ja delete-operaattoreita. (7.)

Kuvassa 9 käytetään kuvassa 8 olevia luokkia. Ajan mittaamiseen käytetään standardikirjastosta löytyvää clock-funktiota.

```
21  int main()
22  {
23      int count = 100000000;
24      int t1 = std::clock();
25      for (int i = 0; i < count; ++i)
26      {
27          TestClass1 *ct1 = new TestClass1;
28          delete ct1;
29      }
30      int x1 = std::clock() - t1;
31      std::cout << "Tavallinen new/delete kesto: " << x1 << " ms" << std::endl;
32
33      int t2 = std::clock();
34      for (int i = 0; i < count; ++i)
35      {
36          TestClass2 *ct2 = new TestClass2;
37          delete ct2;
38      }
39      int x2 = std::clock() - t2;
40      std::cout << "Yksinkertainen muisti allas kesto: " << x2 << " ms" << std::endl;
41
42      return 0;
43  }
```

KUVA 9. TestClass1- ja TestClass2-luokan testaus

Riveillä 25–29 tehdään for-silmukassa 100000000 kertaa dynaaminen muistin varaus ja vapautus käyttäen tavallisia new- ja delete-operaattoreita. Riveillä 34–38 on sama koodi kuin riveillä 25–29, mutta nyt käytetään luokkaa, joka käyttää yksinkertaista muistiallasta. (7.)

Kuvassa 10 näkyy testin tulos.

```
Tavallinen new/delete kesto: 41095 ms
Yksinkertainen muistiallas kesto: 4006 ms
Jatka painamalla mitä tahansa näppäintä . . .
```

KUVA 10. Testin tulos

Kuten kuvasta näkyy, testiohjelma oli valmiiksi varatun muistin kanssa noin 10 kertaa nopeampi kuin tavalliset new- ja delete-operaattorit. Muistialtaan käytöllä voidaan siis huomattavasti nopeuttaa ohjelmaa, jonka tarvitsee tehdä useita dynaamisia muistin varauksia ja vapautuksia. (7.)

6 YHTEENVETO

Tässä opinnäytetyön ensimmäisessä osassa esiteltiin C++:n muistinhallintaa ja sitä varten kehitettyjä tekniikoita. Jos aikaa olisi ollut enemmän, työstä olisi helposti saanut laajemmankin, koska muistinhallinta on hyvin laaja käsite. Työssä käsiteltiin vain yhden säikeen muistinhallintaa. Muistinhallinnan tuntemisesta on hyötyä myös muissakin kielissä kuin C++:a ja sen tuntemisen merkitys vain korostuu sitä mukaa, mitä alemman tason koodia kirjoitetaan.

Työn muistialueet-osassa esiteltiin C++:n muistialueet, joilta muuttujien tilanvaraus tapahtuu. Viittausten laskenta -osassa kerrottiin viittausten laskennan perusteet. Neljännessä osassa esiteltiin RAII-tekniikka ja älykkäät osoittimet. Lopuksi esiteltiin muistialtaan toiminta, ja esiteltiin esimerkkiohjelma, joka käyttää yksinkertaista muistiallasta. Esimerkin perusteella todettiin, että muistialtaan käytöllä voidaan nopeutta dynaamista muistia käyttävien ohjelmien toimintaa.

LÄHTEET

1. mycodeschool 2013. Pointers and dynamic memory – stack vs heap. Video. Saatavissa: https://www.youtube.com/watch?v=_8-ht2AKyH4. Hakupäivä 11.5.2015.

2. Qian, Bo 2012. Advanced C++: Resource Acquisition Is Initialization. Video. Saatavissa: https://www.youtube.com/watch?v=ojOUIg13g3I. Hakupäivä 12.5.2015

3. Horton, Ivor 2014. Beginning C++. Yhdysvallat: Apress.

4. Smart Pointers (Modern C++). 2013. Microsoft. Saatavissa: https://msdn.microsoft.com/en-us/library/hh279674.aspx. Hakupäivä 8.4.2015.

5. How to: Create and Use weak_ptr Instances. Microsoft. Saatavissa: https://msdn.microsoft.com/en-us/library/hh279672.aspx. Hakupäivä 14.4.2015

6. DanDanger2000 2006. C++ Memory Pool. Saatavissa: http://www.codeproject.com/Articles/15527/C-Memory-Pool. Hakupäivä 13.5.2015.

7. Deng, Jude 2008. Why to use memory pool and how to implement it. Saatavissa: http://www.codeproject.com/Articles/27487/Why-to-use-memory-pool-and-how-to-implement-it. Hakupäivä 13.5.2015.

Sami Varanka

C++-LIITÄNNÄISTEN TOTEUTTAMINEN

C++-LIITÄNNÄISTEN TOTEUTTAMINEN

Sami Varanka
Opinnäytetyö, toinen osa
Kevät 2016
Tietotekniikan koulutusohjelma
Oulun ammattikorkeakoulu

SISÄLLYS

1 JOHDANTO

Tietokoneohjelman toteuttaminen on hyvä jakaa moduuleihin, joita on sitten helppo vaihtaa tarvittaessa. Esimerkiksi grafiikan piirtämiseen Windowsin puolella voidaan käyttää moduulia, joka käyttää DirectX-grafiikkakirjastoa. Jos kuitenkaan DirectX ei ole saatavilla, voi sovellus ottaa käyttöön OpenGL-kirjaston moduulia vaihtamalla. Nämä moduulit toteutetaan yleensä DLL-kirjastoina.

Tässä opinnäytetyöni toisessa osassa on tarkoitus käsitellä liitännäisten toteuttamista tietokonesovelluksiin. Liitännäiset toteutetaan DLL-kirjastoina, jolloin ne voidaan lisätä sovellukseen ajon aikana.

Opinnäytetyössä ei ole tarkoitus käsitellä staattisesti linkitettävää kirjastoa. Työssä käsitellään vain Windows-ympäristöä. Työssä käytetään Microsoft Visual Studio 2013 -kehitysympäristöä.

Opinnäytetyön osana laajennetaan Bjarne Stroustrupin kirjasta ”Programming: Principles and Practice using C++” löytyvää laskinsovellusta liitännäisillä, jotka toteutetaan DLL-moduuleina.

Valitsin tämän aiheen, koska minua kiinnostaa sovelluksien toiminnollisuuden jakaminen osiin, joiden ei tarvitse olla saatavilla, kun varsinainen sovellus kootaan. Tämän takia käyttäjä voi itsekin toteuttaa ominaisuuksia sovellukseen, jos sovelluksen kehittäjä on sen mahdollistanut.

2 LINKITETTÄVÄ KIRJASTO

Tietotekniikassa ohjelmakoodia kirjoittaessa koodia voidaan uudelleen käyttää jakamalla se linkitettäviin kirjastoihin. Kirjastot jaetaan linkittämistavan mukaan kahteen eri tyyppiin. Näitä tyyppejä ovat staattisesti ja dynaamisesti linkitettävät kirjastot.

Staattisesti linkitettävä kirjasto yhdistetään sovellukseen koonnin aikana linkittämisvaiheessa ja kirjaston tarjoama koodi on aina sovelluksen exe-tiedoston mukana. Kuitenkin joissain tapauksissa sovellukset käyttävät samaa kirjastoa ja silloin staattisesti linkitettävien kirjastojen käyttö kasvattaa turhaa sovelluksen kokoa. Näissä tapauksissa voidaan käyttää dynaamisesti linkitettävää jaettua kirjastoa. (1.)

Windowsin puolella jaettuja kirjastoja kutsutaan nimellä DLL (Dynamic Link Library). Dynaamisesti linkitettävien kirjastojen käyttö helpottaa sovelluksen modularisointia ja koodin uudelleenkäyttöä, tehostaa muistinkäyttöä ja pienentää sovelluksen kokoa. Tämän takia ohjelma latautuu nopeampaa, toimii nopeampaa ja vie vähemmän kiintolevytilaa tietokoneella. (1.)

2.1 DLL-moduuli

Dynaamisesti linkitettävien kirjastojen käyttö jakaa sovelluksen moduuleihin. Näitä moduuleja voidaan kehittää irrallaan varsinaisesta ohjelmasta, ja kun moduuliin tehdyt muutokset ovat valmiit, sovelluksen entinen moduuli voidaan korvata uudella päivitetyllä moduulilla. DLL-moduuleina toteutetaan kirjastot, joita useammat sovellukset käyttävät. Jos kirjastot, joita useammat sovellukset käyttävät, toteutettaisiin staattisesti linkitettävinä kirjastoina, menisi paljon kiintolevytilaa ja muistia hukkaan. Myös kirjaston päivittyessä pitäisi jokainen kirjastoa käyttävä sovellus uudelleen linkittää.

2.1.1 Tuontikirjasto

Tuontikirjasto (import library) sisältää kaikki DLL:n paljastamat symbolit. Tuontikirjasto on .lib-päätteinen staattisesti linkitettävä kirjasto. Sitä tarvitaan, kun halutaan käyttää DLL:n latauksen aikaista linkitystä (Load-Time Link). Tällöin DLL:n lataaminen tapahtuu sovelluksen alustuksen yhteydessä ennen sovelluksen main-funktion kutsua. Toinen vaihtoehto on käyttää ajonaikaista linkitystä (Run-Time Link).

2.1.2 DLL:n aloituskohta

DLL:n aloituskohta (entry point) on valinnainen funktio. Sitä ei ole pakko toteuttaa, mutta jos sen jättää toteuttamatta, se lisätään automaattisesti. Kun järjestelmä aloittaa tai lopettaa prosessin tai säikeen, se kutsuu aloituskohtaa jokaiselle DLL:lle käyttäen prosessin ensimmäistä säiettä. (2.)

DLL:n aloituskohdassa voi tehdä yksinkertaisia alustustoimintoja, esimerkiksi estää sen kutsumisen DisableThreadLibraryCalls-funktiolla, kun DLL:n ladannut prosessi aloittaa uuden säikeen. Jos DLL:n alustaminen vaatii jotain suurempia toimenpiteitä, olisi parempi tehdä erillinen alustusfunktio. (2; 3.)

Kuvassa 1 näkyy aloituskohdan esittely (prototype).

```
BOOL WINAPI DllMain(
    _In_ HINSTANCE hinstDLL,
    _In_ DWORD     fdwReason,
    _In_ LPVOID    lpvReserved
    );
```

KUVA 1. DLL:n aloituskohdan esittely (2)

Kuten kuvasta näkyy funktio palauttaa 1 tai 0. BOOL on alias int-tyypille, TRUE on makro-määrittely 1:lle ja FALSE on määrittely 0:lle.

WINAPI on puolestaan makro-määrittely kutsutavalle (calling convention) __stdcall. Kutsutapa kertoo kääntäjälle, kuinka argumentit ja paluuarvot liikkuvat

funktion ja kutsujan kesken. Visual C/C++ -kääntäjä ymmärtää muitakin kutsutapoja, kuten __cdecl, joka on peruskutsutapa C-kielen funktioille. (4.)

Jokaisen DllMain-funktion parametrin tyypin edessä oleva _In_ on annotaatio annettaville parametreille. Se kuuluu Microsoftin SAL-kieleen (source-code annotation language), jolla voidaan kertoa kääntäjälle ja lukijalle, kuinka funktio käyttää parametrejaan. (5; 6.)

DLL:n aloituskohdan ensimmäinen parametri on kahva (handle) siihen DLL-moduuliin, jonka aloituskohtaa kutsutaan. Funktion toinen parametri on kutsun syy, joka voi olla jokin seuraavista:

- DLL_PROCESS_ATTACH
- DLL_PROCESS_DETACH
- DLL_THREAD_ATTACH
- DLL_THREAD_DETACH.

Toisen parametrin ollessa DLL_PROCESS_ATTACH kolmas parametri on NULL, jos käytetään ajonaikaista linkitystä, ja eri kuin NULL käytettäessä latauksen aikaista linkitystä. Jos toisena parametrina on DLL_PROCESS_DETACH, kolmas parametri on NULL, jos FreeeLibrary-funktiota on kutsuttu, tai DLL:n lataus epäonnistuu. Jos kolmas parametri ei ole NULL ja toinen parametri on DLL_PROCESS_DETACH, prosessi on suoriutunut loppuun asti. (2.)

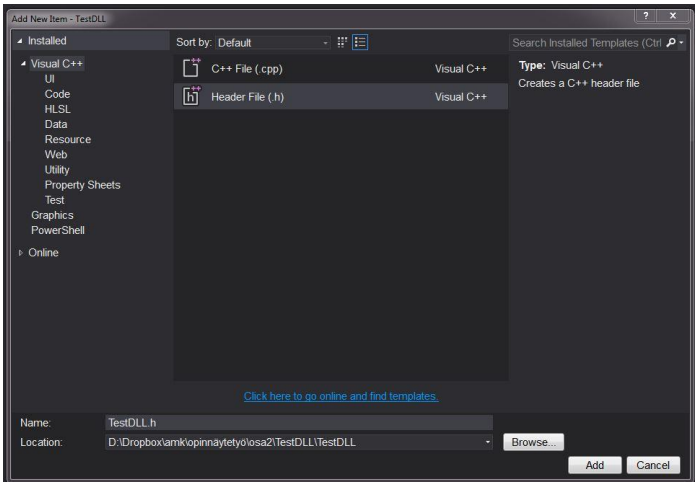
2.1.3 Muistin varaaminen ja vapauttaminen

Jos DLL:n pitää hallita muistia dynaamisesti, täytyy varaaminen ja vapauttaminen tapahtua käyttämällä samaa muistinvaraajaa (memory allocator). Jos DLL-moduuli varaa muistia ja jättää käyttäjän vastuulle muistin vapauttamisen, käyttäjä saattaa yrittää vapauttaa muistin käyttäen väärää muistinvaraajaa. Tämän takia on parasta, että muistin vapauttaminen tapahtuu samassa moduulissa kuin missä varaaminenkin.

2.2 DLL:n toteuttaminen

Seuraavassa esitellään DLL:n toteuttaminen C++-ohjelmointikielellä Windows-ympäristössä käyttäen Microsoft Visual Studio 2013 -kehitysympäristöä. Tässä esitelty tapa ei toimi muilla kääntäjä versioilla kuin millä DLL on koottu. DLL:n toteutus alkaa samalla tavalla kuin minkä tahansa muunkin sovelluksen toteutus. Ensiksi valitaan file-valikosta new project ja sieltä Win32 console application. Sen jälkeen avautuvasta valikosta valitaan projektin tyypiksi DLL. Valitaan lisävalinnoista tyhjä projekti (empty project) ja sitten painetaan valmis (finish).

Lisätään projektiin uusi otsikkotiedosto (header file). Kuvassa 2 näkyy valikko, josta voi lisätä uuden tiedoston.



KUVA 2. Tiedoston lisääminen projektiin

Kuvassa 3 näkyy DLL:n otsikkotiedosto. Tiedostoon on vain kirjoitettu olennaisin koodi. Tässä esitelty DLL paljastaa yhden luokan ja yhden funktion.

```
1  #pragma once
2
3  #ifdef TESTDLL_EXPORTS
4  #define TESTDLL_API __declspec(dllexport)
5  #else
6  #define TESTDLL_API __declspec(dllimport)
7  #endif
8
9  class TESTDLL_API TestClass{
10 public:
11     TestClass();
12     void test();
13 };
14
15 TESTDLL_API void foo();
```

KUVA 3. DLL:n otsikkotiedosto

Ensimmäisellä rivillä #pragma once ilmoittaa esikääntäjälle, että tiedoston sisältö lisätään vain kerran käännettäväksi. Riveillä 3–7 ilmoitetaan esikääntäjälle, että jos symboli TESTDLL_EXPORTS on määritelty, TESTDLL_API-symbolilla merkatut symbolit paljastetaan DLL:stä. Muussa tapauksessa DLL:stä tuodaan merkatut symbolit. (7.)

Kuvassa 4 näkyy TestDLL:n lähdekoodi (source code). Kuten kuvasta näkyy, se ei ole mitenkään erilaista verrattuna tavallisen exe:n lähdekoodiin.

```
1  #include<iostream>
2  #include"TestDLL.h"
3
4  TestClass::TestClass() {
5
6  }
7
8  void TestClass::test() {
9      std::cout << "Hello this is test" << std::endl;
10 }
11
12 void foo() {
13     std::cout << "foobar" << std::endl;
14 }
15
16
```

KUVA 4. DLL.n lähdekoodi

Kuvassa 5 näkyy DLL:n aloituskohdan määrittely. Tätä funktiota ei ole pakko lisätä.

```
#include <windows.h>
#include <cstdio>

BOOL WINAPI DllMain(
    _In_ HINSTANCE hinstDLL,
    _In_ DWORD     fdwReason,
    _In_ LPVOID    lpvReserved
    ){
    switch (fdwReason){
    case DLL_PROCESS_ATTACH:
        printf("DLL process attached\n");
        break;
    case DLL_PROCESS_DETACH:
        printf("DLL process detached\n");
        break;
    case DLL_THREAD_ATTACH:
        printf("DLL thread attached\n");
        break;
    case DLL_THREAD_DETACH:
        printf("DLL thread detached\n");
        break;
    }
    return TRUE;
}
```

KUVA 5. DLL:n aloituskohta

2.2.1 Nimien koristelu

Jotkin ohjelmointikielet tukevat funktioiden ylikuormittamista (function overloading). Tämä tarkoittaa sitä, että voidaan määritellä monta samannimistä funktiota. Funktiot erotetaan toisistaan parametrien tyypin ja määrän avulla.

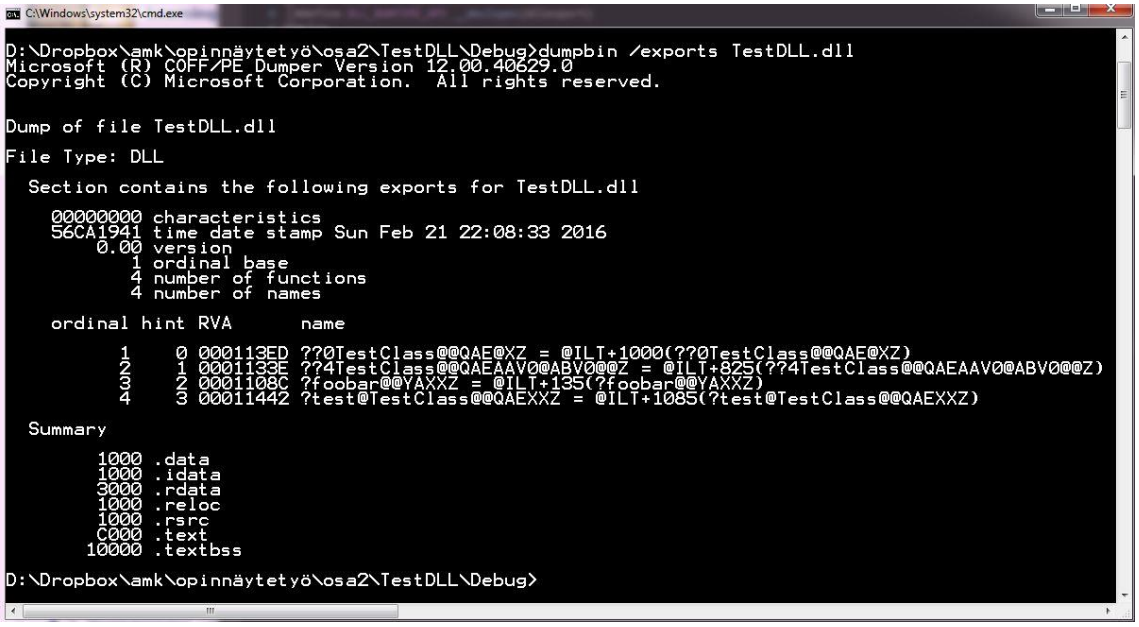
Linkittäjän pitää jotenkin erottaa ylikuormitetut funktiot toisistaan tätä varten kääntäjä tekee nimien koristelua (name decoration, name mangling). Nimien koristelu tarkoittaa sitä, että käännösvaiheessa kääntäjä generoi funktioille omat symboliset nimet. Näiden nimien avulla linkittäjä osaa yhdistää funktiokutsut oikeisiin funktiomäärittelyihin.

C++-ohjelmointikielessä nimien koristelu aiheuttaa ongelmia DLL:n paljastetuiden funktioiden kanssa, koska C++:ssa ei ole standardoitua tapaa, miten funktiot pitäisi koristella, joten jokaisella C++-kääntäjällä on erilainen tapa

koristella funktioita. Tämän takia toisella kääntäjällä käännetty DLL ei toimi toisella kääntäjällä.

Ohjelmoijan täytyy tietää koristellun funktion nimi esimerkiksi silloin, kun käytetään DLL:n ajonaikaista linkittämistä. DLL:n paljastamien funktioiden nimet voidaan saada selville Visual Studion mukana tulevan Dumpbin-työkalun avulla. Dumpbin on Windowsin komentokehotteella ajettava sovellus. Sovellusta pääsee käyttämään valitsemalla Visual Studiossa ensiksi ylävalikosta Tools ja sieltä Visual Studio Command Prompt. Sitten navigoidaan kansioon, jossa DLL sijaitsee, ja kirjoitetaan kehotteeseen dumpbin /exports <tarkasteltava DLL>.

Kuvassa 6 näkyy aikaisemmin tehdyn TestDLL:n paljastamat symbolit.

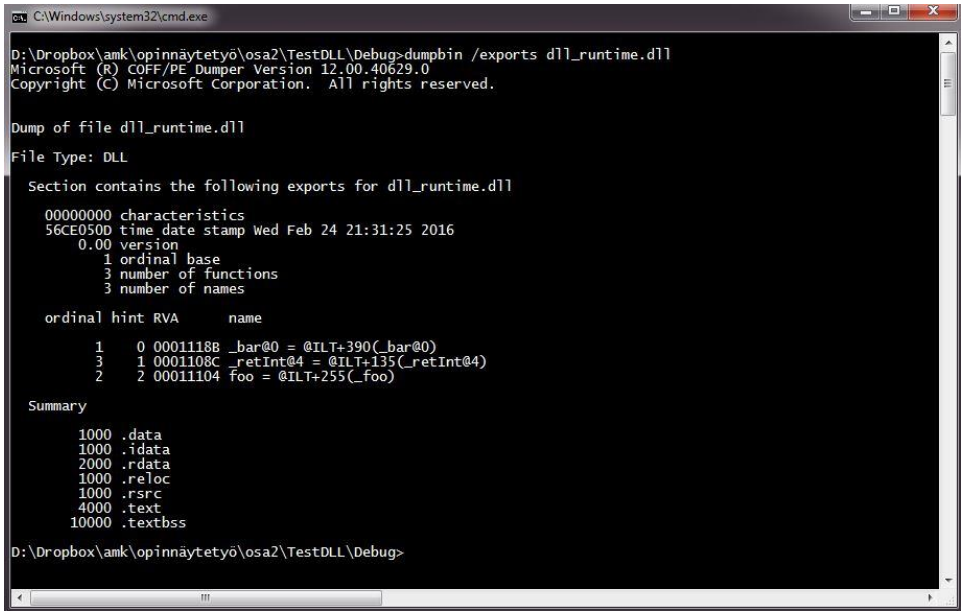


KUVA 6. TestDLL:n paljastamat symbolit

Kuvasta voi nähdä, että funktion foobar koristeltu nimi on ”?foobar@@YAXXZ”, kun taas lähdekoodissa se on esitelty ”void foobar()”.

Kun toteutetaan DLL-moduuleja, joita on tarkoitus käyttää muillakin kääntäjillä, on parempi toteuttaa paljastettavat funktiot C-kielellä. C-kielessä ei voi ylikuormittaa funktioita, eikä siinä siksi, kutsutavasta riippuen, ole suurta tarvetta

nimienkoristelulle. Kuvassa 7 on erään toisen DLL:n paljastamat symbolit tulostettu Dumpbin-työkalulla.



KUVA 7. C-tyylisten funktioiden koristelu

DLL paljastaa kolme C-tyylistä funktiota, joiden nimet ovat bar, retInt ja foo. Funktioiden esittelyt ovat

- void foo()
- void bar()
- int retInt(int).

Funktioista bar ja retInt käyttävät samaa WINAPI-kutsutapaa, kun taas foo käyttää WINAPIV-kutsutapaa, joka on makro-määrittely C-kielen peruskutsutavalle __cdecl. WINAPI-kutsutavassa, joka on __stdcall, funktion koristeltumuoto muodostetaan lisäämällä alaviiva funktion nimen eteen ja perään lisätään @-merkki, jonka jälkeen tulee numero. Tämä numero kertoo, kuinka monta tavua parametrit tarvitsevat tilaa yhteensä. Funktion nimeä ei koristella käytettäessä C-kielen peruskutsutapaa.

2.2.2 Kokoaminen

Kun kaikki tarvittava koodi on saatu kirjoitettua, valitaan yläpalkista build-valikosta kohta build solution. Tämän jälkeen kääntäjä kääntää lähdekooditiedostot. Kääntämisen jälkeen käännetyt lähdekooditiedostot menevät linkittäjälle, joka linkittää tiedostot. Kokoamisprosessin tuloksena saadaan kaksi tiedostoa varsinainen .dll-tiedosto sekä tuontikirjasto.

3 LIITÄNNÄINEN

Jos sovelluksen kehittäjät ovat sen sallineet, sovelluksen toimintaa voidaan laajentaa liitännäisillä (plug-in, addon, extension). Liitännäinen on jäsennelty paketti koodia ja dataa, joka lisää toimintoja sovellukseen. Monet sovellukset mahdollistavat käyttäjien itse toteuttamien liitännäisten käytön, esimerkiksi Unity3D ja Visual Studio. (8.)

3.1 Toteutus DLL-moduulina

Liitännäinen voidaan toteuttaa erillisenä DLL-moduulina, joka toteuttaa liitännäisen käyttöliittymän (interface) esittelemät funktiot. Tällöin sovelluksen kehittäjän täytyy joko kertoa dokumentaatiossa, millaiset funktiot moduulin täytyy toteuttaa, tai jakaa otsikkotiedosto, jossa esitellään toteutettavat funktiot.

3.2 Toteutus skriptillä

Toinen tapa, jolla liitännäisten toteuttaminen voidaan mahdollistaa, on kirjoittaa erillisiä skriptitiedostoja (script). Esimerkiksi Unity3D:ssä käyttäjä voi kirjoittaa skriptin, paketoida sen erilliseen asset-pakettiin (asset package) ja viedä sen Unityn asset-kauppaan (asset store). Unityn asset-kaupasta löytyy monia käyttäjien toteuttamia liitännäisiä. Esimerkiksi iTween on PixelPlacementin tehokas ja helppokäyttöinen liitännäinen objektien liikkeen animoimiseen. (9.)

4 LASKIN

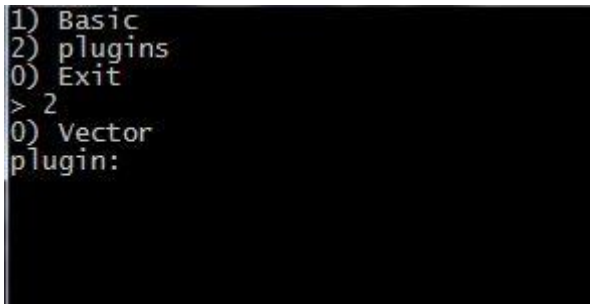
Tästä alkaa opinnäytetyön toisen osan ohjelmointityön esittely. Työssä laajennettiin Bjarne Stroustrupin kirjasta ”Programming: Principles and Practice Using C++” löytyvää laskinsovellusta lisäämällä siihen mahdollisuus laskea vektorilaskuja. Koska työssä ei ollut tarkoitus toteuttaa täydellistä vektorilaskinta, laskimella pystyy laskemaan vain yksinkertaisia peruslaskuja.

4.1 Toiminta

Kun laskimen käynnistää, ensimmäisenä näkyviin tulee laskimen perusvalikko. Valikosta voi valita haluaako käyttää peruslaskinta vai jotain liitännäistä. (Kuva 8.)

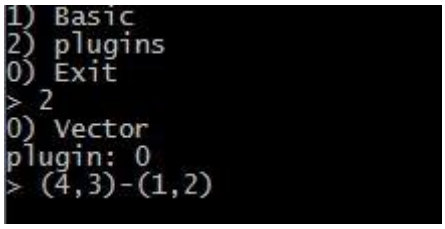
KUVA 8. Laskimen perusvalikko

Peruslaskimella voi laskea yhteen- ja vähennyslaskuja sekä kerto- ja jakolaskuja. Plugins-valikosta pääsee käyttämään laskimen löytämiä liitännäisiä. (Kuva 9.)



KUVA 9. Laskimen plugins-valikko

Seuraavaksi syötetään haluttua liitännäistä vastaava numero. Sitten kirjoitetaan laskettava lauseke. (Kuva 10.)



KUVA 10. Vektorilaskin

Kuvassa 11 näkyy laskimen antama tulos.

KUVA 11. Vektorilaskun tulos

Jos sovellus ei löydä yhtään liitännäistä plugins-kansiosta, sovellus tulostaa tekstin ”no plugins”. (Kuva 12.)

KUVA 12. Ei liitännäisiä

4.2 Toteutus

Laskimen perustoimintojen toteutus on kerrottu Bjarne Stroustrupin kirjassa ”Programming: Principles and Practice using C++” luvuissa 6.3–7.3. Laskimeen mahdollistettiin liitännäisten käyttö lisäämällä siihen PluginSystem-luokka. (10.)

PluginSystem toteutettiin staattisesti linkitettävänä kirjastona. PluginSystem-luokan käyttämät liitännäiset toteutettiin dynaamisesti linkitettävinä kirjastoina. Niiden sisäinen toiminta toteutettiin C++-kielellä, mutta ne paljastavat itsestään vain yhden C-kielisen getPlugin-funktion, joka palauttaa osoittimen IPlugin-käyttöliittymään (interface). IPlugin on C-käyttöliittymä (C-interface) DLL:ssä olevaan luokkaan. (Kuva 13.)

```
struct IPlugin {

    virtual void use(const char *exp) = 0;

    virtual const char *getName() = 0;

    virtual void release() = 0;
};
```

KUVA 13. Liitännäisen käyttöliittymä

Kuvassa 14 näkyy getPlugin-funktion esittely.

```
EXTERN_C VECCALCDLL_API IPlugin* WINAPIV getPlugin(VOID);
```

KUVA 14. getPlugin-funktion esittely

EXTERN_C kertoo kääntäjälle, että tämä on C-kielinen funktio. VECCALCDLL_API kertoo, että tämä funktio paljastetaan DLL:stä. Funktio palauttaa osoittimen IPlugin-käyttöliittymään. WINAPIV kertoo, että käytetään C-kielen peruskutsutapaa __cdecl.

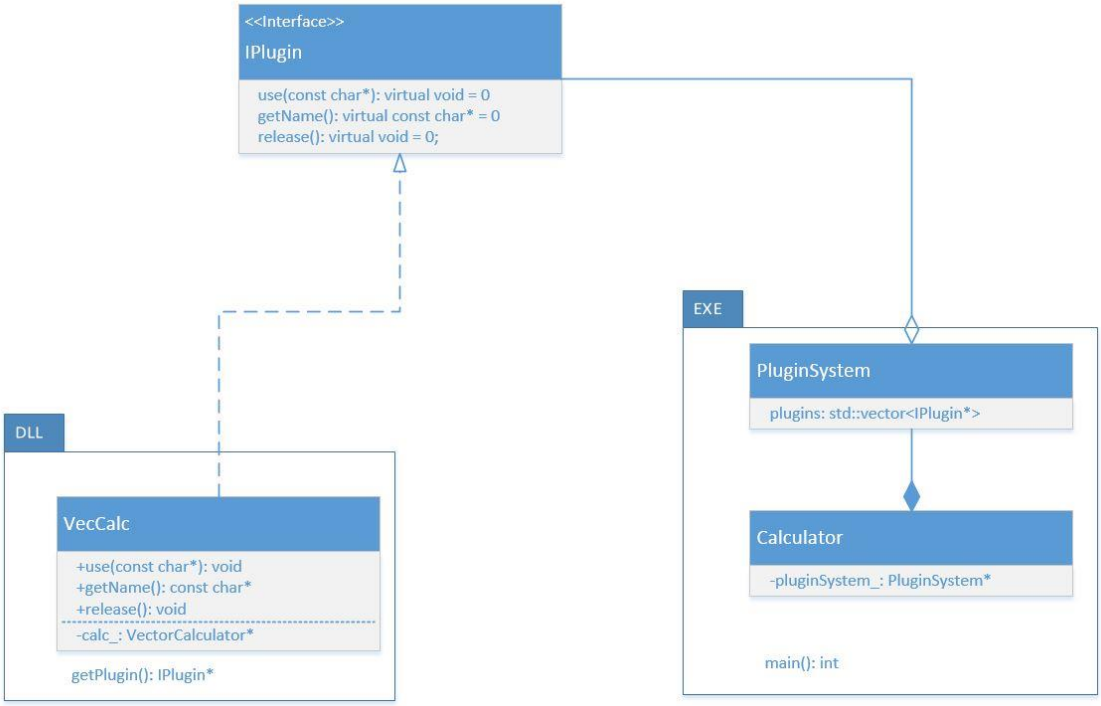
Kuvassa 15 näkyy IPlugin-käyttöliittymän toteuttavan VecCalc-luokan esittely. VecCalc on DLL:n sisäinen luokka, jota käytetään IPlugin-käyttöliittymän kautta.

```
class VecCalc : public IPlugin {
public:
    VecCalc();

    void release();
    void use(const char *exp);
    const char* getName();
private:
    VectorCalculator *calc_;
};
```

KUVA 15. VecCalc-luokan esittely

Kuvasta 16 nähdään, että PluginSystem käyttää liitännäisiä IPlugin-käyttöliittymän kautta.



KUVA 16. Liitännäisen käyttö

Käynnistyessään Calculator-luokka luo osoittimen PluginSystem-luokan olioon. Olio kutsuu PluginSystem-luokasta löytyvää loadPlugins-metodia, joka lataa kaikki löytyvät liitännäiset sovelluksen plugins-kansiosta. Kuvassa 17 näkyy loadPlugins-metodin toteutus.

```
3  #include "PluginSystem.h"
4  #include "IPlugin.h"
5
6  void PluginSystem::loadPlugins() {
7
8
9      std::vector<std::string> foundlibs;
10
11     getFileNames(foundlibs);
12
13     for (size_t i = 0; i < foundlibs.size(); ++i) {
14
15         LPCSTR path = ".\\plugins\\";
16         std::string libName = foundlibs[i];
17         std::string lib = std::string(path) + libName;
18
19         HINSTANCE mod = LoadLibraryA(lib.c_str());
20
21         if (!mod) {
22             std::cerr << "Error: could not load library " << libName.c_str() << std::endl;
23
24             for (HINSTANCE mod : mods_) {
25                 FreeLibrary(mod);
26             }
27             return;
28         }
29
30         ObjFunc objFunc = (ObjFunc)GetProcAddress(mod, "getPlugin");
31
32         IPlugin *plugin = objFunc();
33
34         mods_.push_back(mod);
35         plugins_.push_back(plugin);
36     }
37 }
```

KUVA 17. Liitännäisten lataus

Riveillä 3–4 lisätään PluginSystem-luokan ja IPlugin-käyttöliittymän otsikkotiedostot. Rivillä 11 kutsutaan getFileNames-metodia, joka täyttää foundlibs-listan kaikilla plugins-kansiosta löytyvillä .dll-loppuisilla tiedostonimillä. Rivillä 19 kutsutaan LoadLibrary-funktiota, joka lataa DLL-moduulin sille parametrina annetusta polusta. LoadLibrary palauttaa kahvan ladattuun moduuliin. Rivillä 30 noudetaan osoitin DLL:n paljastamaan funktioon GetProcAddress-funktiolla. GetProcAddress-funktiolle annetaan ensimmäisenä parametrina kahva moduulin, josta funktiota etsitään. Toinen parametri on funktion koristeltu nimi, joka tässä tapauksessa on getPlugin, koska getPlugin-funktio on määritelty käyttämään C-kielen peruskutsutapaa (__cdecl).

GetProcAddress-funktion palauttama funktio-osoitin pitää vielä tyyppimuuntaa (type cast) oikean tyyppiseksi funktio-osoittimeksi. Nimi ObjFunc on vain alias getPlugin-funktion osoitintyypille. Kun getPlugin-funktioon on noudettu osoitin, sitä kutsutaan osoittimen kautta rivillä 32 ja se palauttaa osoittimen liitännäisen käyttöliittymään. Lopuksi riveillä 34–35 moduulin kahva ja liitännäisen käyttöliittymän osoitin talletetaan.

Kun käyttäjä haluaa käyttää jotain liitännäistä, Calculator-luokan olio kutsuu PluginSystem-luokan availablePlugins-metodia. Metodi käy läpi listan, johon on talletettu IPlugin-osoittimia ja tulostaa jokaisen nimen. Metodi tulostaa myös jokaisen indeksinumeron.

Kun käyttäjä on valinnut haluamansa liitännäisen käyttäjä syöttää laskutoimituksen, jonka jälkeen Calculator-luokan olio kutsuu PluginSystem-luokan usePlugin-metodia. Metodille annetaan kaksi parametria. Ensiksi annetaan indeksinumero, mistä indeksistä liitännäisen käyttöliittymäosoitin löytyy. Toinen parametri on laskulauseke, joka halutaan ratkaista. (Kuva 18.)

```
void PluginSystem::usePlugin(size_t i, std::string &str) {
    plugins_[i]->use(str.c_str());
}
```

KUVA 18. usePlugin-metodi

PluginSystem-luokan usePlugin-metodi kutsuu IPlugin-osoitinlistasta löytyvän käyttöliittymän kautta liitännäisen use-metodia. Metodille annetaan parametrina laskulauseke.

Kun ohjelma lopetetaan, kutsutaan Calculator-luokan shutDown-metodia, joka puolestaan kutsuu PluginSystem-luokan shutDown-metodia. PluginSystem-luokan shutDown-metodi käy läpi koko IPlugin-osoitinlistan ja kutsuu jokaisen release-metodia. Lopuksi shutDown-metodi käy läpi listan, johon on talletettu ladatut DLL-moduulit, ja vapauttaa ne.

5 YHTEENVETO

Tässä opinnäytetyön toisessa osassa oli tarkoitus tutkia liitännäisten toteuttamista DLL-moduuleina. Työssä laajennettiin Bjarne Stroustrupin kirjasta ”Programming: Principles and Practice using C++” löytyvää laskinta lisäämällä siihen liitännäinen, joka mahdollistaa yksinkertaiset vektorilaskut.

Työ onnistui hyvin. Laskimeen toteutettiin PluginSystem-luokka, joka pystyy käyttämään plugins-kansiosta löytyvää liitännäistä. Liitännäinen toteutettiin DLL-moduulina. PluginSystem käyttää ajonaikaista linkittymistä DLL-moduuliin.

Mielestäni työn ohjelmointiosuus oli sopivan suuruinen ja sitä oli mukava tehdä. Työtä tehdessä opin lisää dynaamisesti linkitettävien kirjastojen toiminnasta ja niiden toteuttamisesta. Opin myös eri kutsutavoista ja kääntäjien tekemästä funktioiden nimien koristelusta. Lisäksi opin käyttöliittymien hyödyistä, kuinka niitä voidaan hyödyntää käytettäessä DLL-moduuleja.

LÄHTEET

1. What is a DLL? 2016. Microsoft support. Saatavissa: https://support.microsoft.com/en-us/kb/815065. Hakupäivä 18.2.2016.

2. DllMain entry point. 2016. Microsoft Developer technologies. Saatavissa: https://msdn.microsoft.com/en-us/library/windows/desktop/ms682583(v=vs.85).aspx. Hakupäivä 23.2.2016.

3. DisableThreadLibraryCalls function. 2016. Microsoft Developer technologies. Saatavissa: https://msdn.microsoft.com/en-us/library/windows/desktop/ms682579(v=vs.85).aspx. Hakupäivä 23.2.2016.

4. Argument Passing and Naming Conventions. 2013. Microsoft Developer Network. Saatavissa: https://msdn.microsoft.com/en-us/library/984x0h58.aspx. Hakupäivä 23.2.2016.

5. Annotating Function Parameters and Return Values. 2013. Microsoft Developer Network. Saatavissa: https://msdn.microsoft.com/en-us/library/hh916382.aspx. Hakupäivä 24.2.2016.

6. Understanding SAL. 2013. Microsoft Developer Network. Saatavissa: https://msdn.microsoft.com/en-us/library/hh916383.aspx. Hakupäivä 24.2.2016

7. Walkthrough: Creating and using a Dynamic Link Library (C++). 2013. Microsoft Developer Network. Saatavissa: https://msdn.microsoft.com/en-us/library/ms235636.aspx. Hakupäivä 19.2.2016.

8. A Plugin-Based Software Production Line Integrated Framework. Zhu, Jun – Yin, Quan – Zhu, Rui – Guo, Changguo – Wang, Huaimin – Wu, Quanyuan 2008. Hubei: International Conference on Computer Science

and Software Engineering 12–14.12.2008. S. 562–565. Saatavissa: http://ieeexplore.ieee.org.ezp.oamk.fi:2048/stamp/stamp.jsp?tp=&arnumber=4722114&tag=1. Hakupäivä 1.3.2016.

9. iTween. Unity - Pixelplacement. Saatavissa: https://www.assetstore.unity3d.com/en/#!/content/84. Hakupäivä 1.3.2016.

10. Stroustrup, Bjarne 2014. Programming: Principles and Practice Using C++, Second Edition. Yhdysvallat: Addison-Wesley Professional.

LIITE 3/1

OULUN AMMATTIKORKEAKOULU

Sami Varanka

MOBIILIPELIN TOTEUTTAMINEN UNITY-PELIMOOTTORILLA SEKÄ PISTEIDEN TALLENNUS JA LATAUS PILVESTÄ

MOBIILIPELIN TOTEUTTAMINEN UNITY-PELIMOOTTORILLA SEKÄ PISTEIDEN TALLENNUS JA LATAUS PILVESTÄ

Sami Varanka
Opinnäytetyö, kolmas osa
Kevät 2017
Tietotekniikan koulutusohjelma
Oulun ammattikorkeakoulu

SISÄLLYS

1 JOHDANTO

Opinnäytetyöni kolmannessa osassa toteutan mobiilipelin Unity-pelimoottorilla. Pelin työnimi on ”Bomb Catcher”. Pelissä on tarkoitus kerätä ylhäältä putoavia pommeja koriin. Kun pelaaja saa yhden pommin kiinni, hän saa yhden pisteen.

Projektin tavoitteena on myös toteuttaa peliin pisteiden hakeminen ja tallennus pilvitietokantaan. Tämän takia täytyy myös toteuttaa yksinkertainen palvelin, joka ottaa yhteytä tietokantaan. Lisäksi pelin koodin rakenteen tulee olla modulaarinen ja moduuleja pitää pystyä vaihtamaan. Projektin versionhallintaan käytetään Gittiä ja BitBuckettia.

Pelin toteutuksessa käytetään Windows-PC:tä, Unity-pelimoottoria sekä C#-ohjelmointikieltä. Pilvitietokanta käyttää mLabin MongoDB-tietokantaa, joka käyttää Microsoftin Azure-pilvialustaa. Palvelin, joka ottaa yhteyttä pilvitietokantaan, toteutetaan NodeJS-ympäristöön ja siinä käytetään JavaScript-ohjelmointikieltä. Palvelin laitetaan Heroku-pilvialustalle, jolloin sitä ei tarvitse erikseen pitää päällä paikallisella kotikoneella.

Tämän työn tilaajana toimii Pertti Heikkilä, mutta työn aihe on itsekeksitty ja esitelty tilaajalle. Valitsin tämän aiheen, koska olen kiinnostunut pelien teosta ja haluan kehittää taitojani Unity-pelimoottorin parissa. Minua kiinnostaa myös verkkoyhteyden välityksellä toimivien moninpeliominaisuuksien toteuttaminen peleihin.

2 UNITY JA MONGODB

2.1 Unity

Unity on Unity Technologiesin tuottama pelimoottori ja pelinkehitysympäristö. Unitylla kehitetty peli voidaan kääntää usealle erilaiselle Unityn tukemalle alustalle. Unity tukee kaikkia suurimpia mobiili-, VR-, työpöytä-, konsoli- ja TV-alustoja sekä WebGL:ää. (1.)

2.1.1 Unityn kehitys

Vuonna 2002 Nicholas Francis ja Joachim Ante alkoivat kehittämään varjostinohjelmointijärjestelmää, jota he molemmat voisivat käyttää omissa pelimoottoreissaan. Jonkin ajan kuluttua he päättivät kuitenkin luopua omista pelimoottoreistaan ja alkoivat yhdessä kehittämään pelimoottoria. Hieman myöhemmin David Helgason liittyi mukaan projektiin kolmanneksi kehittäjäksi. (2.)

Unityn ensimmäinen 1.0.0-versio julkaistiin 6.6.2005. Unityn ensimmäisellä versiolla ei pystynyt vielä luomaan Windows-alustalla toimivia pelejä, vaan tuki tälle lisättiin versioon 1.1, joka julkaistiin 23.8.2005. (3; 4.)

Ennen vuotta 2009 täytyi omistaa Mac-tietokone, jos halusi kehittää pelejä Unitylla. Tähän saatiin muutos 2.5-version julkaisun myötä, kun Unityn työkalut saatiin toimimaan Windows-ympäristössä. (5.)

Tällä hetkellä Unityn uusin vakaa versio on 5.5. Versio 5.6 on tällä hetkellä beetatestauksessa ja sen vakaa versio on tarkoitus julkaista maaliskuussa 2017. Unityn versio 5.6 tulee olemaan viimeinen Unity 5 -versio ja sen jälkeen siirrytään Unity 2017 -versioon. (6; 7.)

2.1.2 Toiminta

Unityssa pelin tasot koostuvat näkymistä (scene). Näkymä on vain tyhjä tila. Näkymään lisätään peliobjekteja (gameobject). Peliobjektiin voidaan lisätä

komponentteja (component). Komponentit antavat toiminnollisuuden peliobjekteille. Kun Unityssa luo uuden peliobjektin, siihen lisätään automaattisesti paikan antava transform-komponentti. Ilman transform-komponenttia peliobjektin paikkaa ei voi määrittää näkymässä.

Unityssa on useita valmiita komponentteja, joilla kehittäjä voi koostaa monenlaisia peliobjekteja. Kuitenkaan valmiina olevat komponentitkaan eivät yleensä riitä, joten kehittäjä voi itse tehdä lisää komponentteja kirjoittamalla skripteja (script). Skriptit ovat kooditiedostoja, joita voidaan kirjoittaa joko C#- tai JavaScript-ohjelmointikielellä.

Jos näkymään halutaan lisätä pelaajan liikuttama hahmo, täytyy peliobjektiin lisätä ainakin kaksi komponenttia: komponentti, joka antaa peliobjektille grafiikan, sekä komponentti, joka lukee pelaajan syötteitä näppäimistöltä ja reagoi niihin muuttamalla transform-komponentin arvoja.

2.2 MongoDB

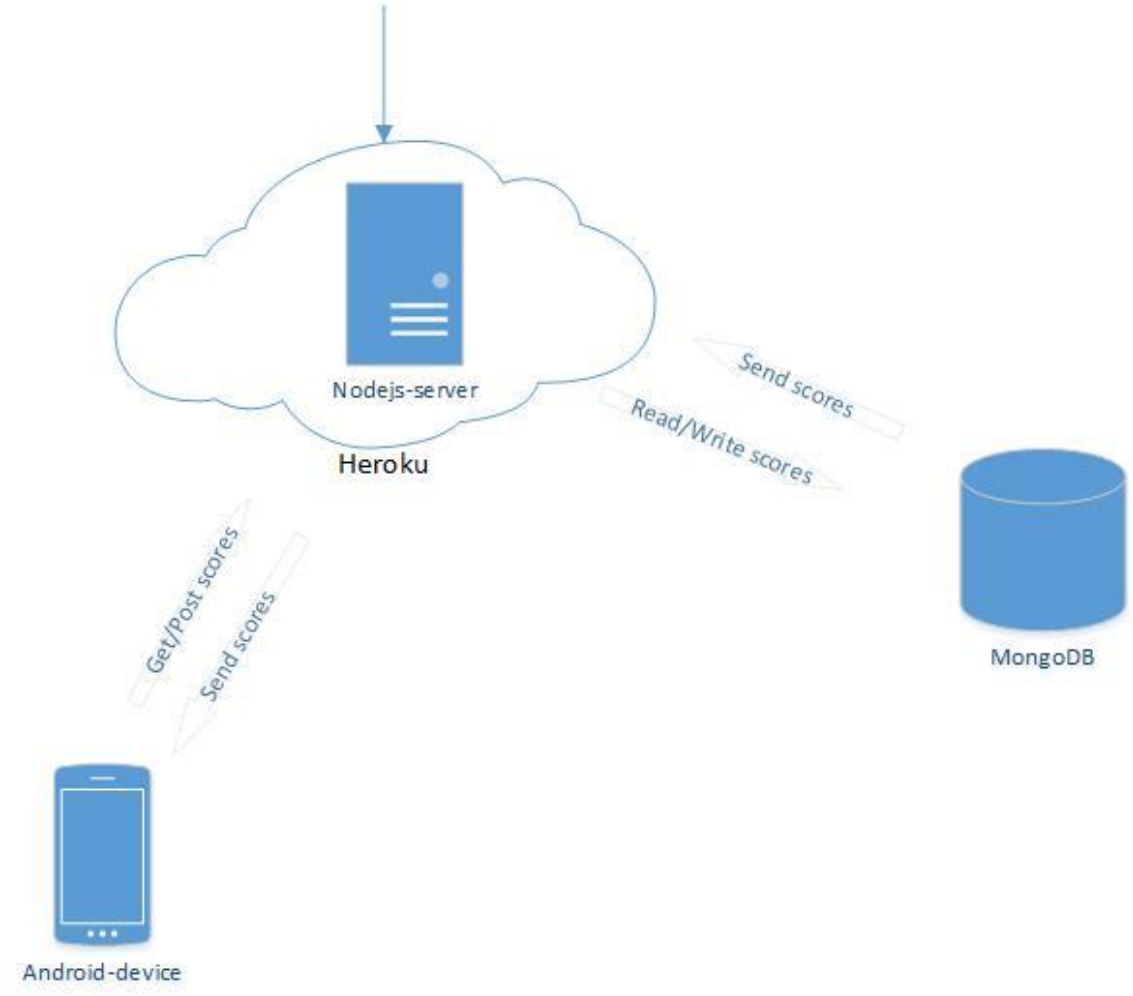
MongoDB on dokumenttipohjainen tietokanta. Se sisältää kokoelmia, jotka sisältävät dokumentteja. Jokainen dokumentti on joukko avain-arvopareja, jossa arvo voi olla merkkijono, numero, totuusarvo, päivämäärä, taulukko tai alidokumentti. Dokumentit ovat ryhmitelty kokoelmiin ja kokoelmat ovat ryhmitelty tietokantoihin. (8.)

3 TOTEUTUS

Tässä luvussa selostetaan pilvitietokantayhteyden toteutus sekä toteutetun pelin kehitys pääpiirteittäin.

3.1 Tietokantayhteys

Tietokantayhteyden toteutuksen ohjeet löytyvät osoitteesta: https://goocreate.com/learn/a-simple-database-driven-high-score-list/. Ohjeita täytyi kuitenkin hieman soveltaa. Kuvassa 1 on esitetty tietokantayhteyden sijoittelukaavio.



KUVA 1. Sijoittelukaavio

Ensiksi toteutettiin NodeJS-palvelin, jonka kautta tehdään kaikki kyselyt mLabin MongoDB-tietokantaan. Kun peli lähettää palvelimelle GET-pyynnön, se hakee tietokannasta viisi parasta tulosta ja lähettää ne pelille JSON-formaatissa. Kuvassa 2 näkyy käsittelijäfunktio (handler function), joka suoritetaan GET-pyynnön tullessa palvelimelle.

```
// GET scores, sorted by 'score'
router.get('/', cors(), function(req, res) {
    console.log('GET scores');

    // Get the database object we attached to the request
    var db = req.db;
    // Get the collection
    var collection = db.get('highscores');
    // Find all entries sort by 'score' ascending
    collection.find({}, { limit: 5, sort: { score: -1 } }, function(err, docs) {
        if(err) {
            console.error('Failed to get scores', err);
            res.status(500).send('Failed to get scores');
        } else {
            res.json(docs);
        }
    });
});
```

KUVA 2. GET-pyyntö

Kun halutaan tallentaa pelaajan saamat pisteet tietokantaan, palvelimelle lähetetään POST-pyyntö, johon liitetään pelaajan nimi ja pisteet. Käsittelijäfunktio, joka suoritetaan, kun palvelimelle tulee POST-pyyntö, näkyy kuvissa 3 ja 4.

```
43  //POST a score
44  router.post("/", cors(), function(req, res) {
45      console.log("POST score");
46      var t_name = req.body.name;
47      var score = Number(req.body.score);
48      if(!(t_name && score)) {
49          console.error("Data formatting error");
50          res.status(400).send("Data formatting error");
51          return;
52      }
53
54      var db = req.db;
55      var collection = db.get("highscores");
56
57      //find players current score
58      //if new score is bigger than collection's score
59      //update score for that player
```

KUVA 3. POST-pyynnön käsittelijän alku

Funktion alussa pyynnön rungosta (body) noudetaan pelaajan nimi ja pisteet. Tiedot ovat merkkijonoja, joten pisteet täytyy muuttaa numeroksi. Kuvan lopussa näkyy, kuinka tietokannasta haetaan highscores-niminen kokoelma (collection), joka sisältää JSON-olioita. Jokaiseen JSON-olioon on talletettu yhden pelaajan nimi ja pisteet sekä MongoDB:n lisäämä id.

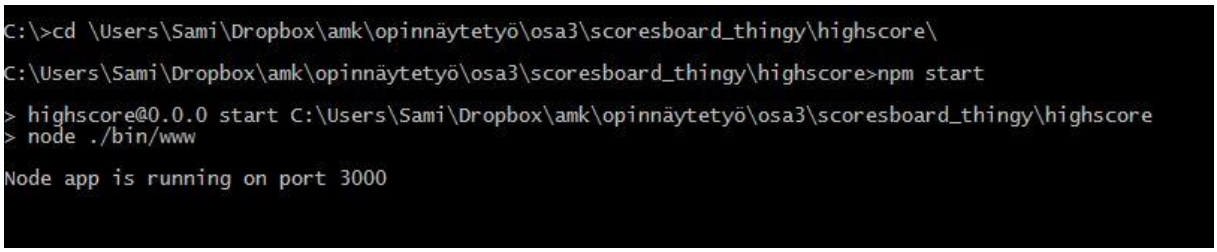
Kuvassa 4 näkyy loput POST-pyynnön käsittelijäfunktiosta.

```
62      collection.find( { name: t_name }, { limit: 1 }, function(err, doc){
63          if(err) {
64              console.error("Error with find");
65          } else {
66              if(doc.length > 0) {
67                  console.log("scoreitem with playername in collection.");
68                  scoreItem = doc[0];
69
70                  console.log(typeof scoreItem.score);
71                  console.log(typeof score);
72                  if(scoreItem.score < score) {
73                      console.log("Score needs to be updated");
74                      console.log("update player's score");
75                      collection.update({ name: t_name }, {
76                              'name': t_name,
77                              'score': score
78                          }, { upsert: true }, function(err, doc) {
79                              if(err) {
80                                  console.error("Updated failed");
81                              } else {
82                                  res.json(doc);
83                              }
84                          }
85                      );
86
87                  }
88              } else {
89                  collection.insert( { 'name': t_name, 'score': score },
90                      function(err, doc) {
91                          if(err) {
92                              res.status(500).send("Error while inserting new scoreitem");
93                          } else {
94                              console.log("success!");
95                              res.json(doc);
96                          }
97                      });
98              }
99          }
100     });
101 });
```

KUVA 4. POST-pyynnön käsittelijän loppu

Koodissa testataan, löytyykö kokoelmasta pelaajan nimellä tallennettu tulos. Jos kokoelmasta löytyy pelaajan nimellä tallennettu tulos, saatua pistemäärää verrataan tallennettuun pistemäärään. Jos saatu pistemäärä on suurempi kuin tallennettu pistemäärä, tallennettu pistemäärä korvataan uudella pistemäärällä. Jos pelaajan nimellä ei löydy yhtään tallennettua tulosta, kokoelmaan lisätään uusi tulos.

Kun palvelin oli saatu valmiiksi, täytyi sitä testata. Palvelinta testattiin paikallisesti, ennen kuin se laitettiin Heroku-pilvialustalle. Kuvassa 5 näkyy, kuinka palvelin käynnistetään paikallisella tietokoneella.



KUVA 5. Palvelimen käynnistys paikallisesti

Ensiksi avattiin komentokehote ja navigoitiin kansioon, jossa bin-kansio sijaitsee. Sitten suoritettiin komento ”npm start”, jolloin palvelin käynnistyi paikallisesti porttiin 3000, joka oli määritetty bin-kansiossa olevassa aloitusskriptissä.

Kun palvelimen toiminta oli varmistettu paikallisesti, se laitettiin Heroku-pilvialustalle. Herokun käyttöön tarvittavat ohjeet löytyivät osoitteesta https://devcenter.heroku.com/articles/getting-started-with-nodejs#introduction. Ohjeita täytyi hieman soveltaa.

3.2 Peli

Valmiissa pelissä on kolme näkymää, joita ovat peli, valikko ja pistetaulu. Pelissä on kolme peliobjektia, joita ei tuhota siirryttäessä näkymästä toiseen. Nämä peliobjektit ovat

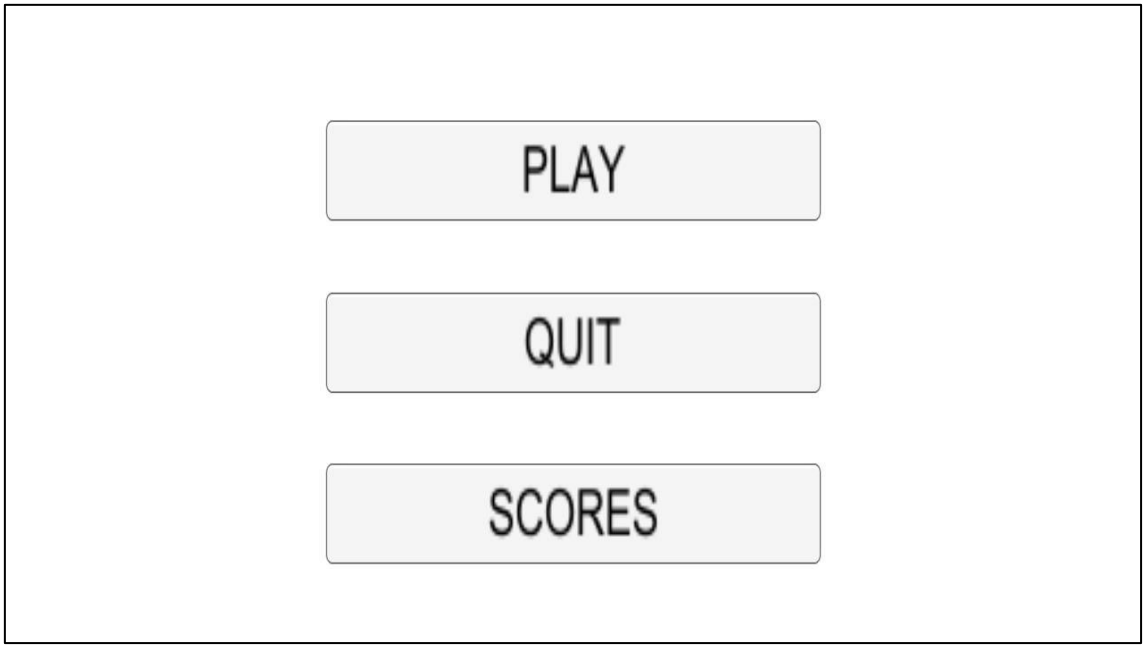
- Mediator
- GameManager
- SceneLoader.

Mediator-peliobjektiin on liitetty Mediator-niminen skripti, jolle muut skriptit voivat ilmoittaa tapahtumista ja Mediator-skripti nostaa vastaavan tapahtuman. Näin saadaan vähennettyä skriptien välisiä viittauksia. GameManager-

peliobjektiin on lisätty GameManager-skripti, joka hallinnoi peliä yleisesti sekä pitää yllä tietoa pisteistä. Näkymien lataamisesta vastaa SceneLoader-skripti, joka on liitetty SceneLoader-peliobjektiin.

3.2.1 Valikkonäkymä

Pelin käynnistyessä ensiksi tulee näkyviin valikko, kuten kuvassa 6.



KUVA 6. Pelin valikko

Valikossa on kolme painiketta. PLAY-painikkeesta pelaaja pääsee pelaamaan peliä, QUIT-painikkeesta pelaaja voi sulkea pelin ja SCORES-painikkeesta pelaaja pääse katsomaan pistetaulua. Jokaisen painikkeen toiminta periaate on sama. Kun painiketta painetaan, kutsutaan painikkeen klikkaustapahtumaan (click event) kytkettyä käsittelijäfunktiota. Käsittelijäfunktio kutsuu Mediator-skriptin painiketta vastaavaa funktiota, joka puolestaan nostaa tapahtuman (raise event), johon SceneLoader-skripti reagoi ja lataa oikean näkymän.

Kuvassa 7 näkyy PLAY-painikkeen klikkaustapahtumaan kytketty käsittelijäfunktio.

```
public void PlayBtn() {
    Debug.Log("Play button clicked");
    Mediator.Instance.OnPlayButtonClick();
}
```

KUVA 7. PLAY-painikkeen käsittelijäfunktio

Käsittelijäfunktio kutsuu Mediator-skriptin OnPlayButtonClick-funktiota. Kuvassa 8 näkyy OnPlayButtonClick-funktion toteutus.

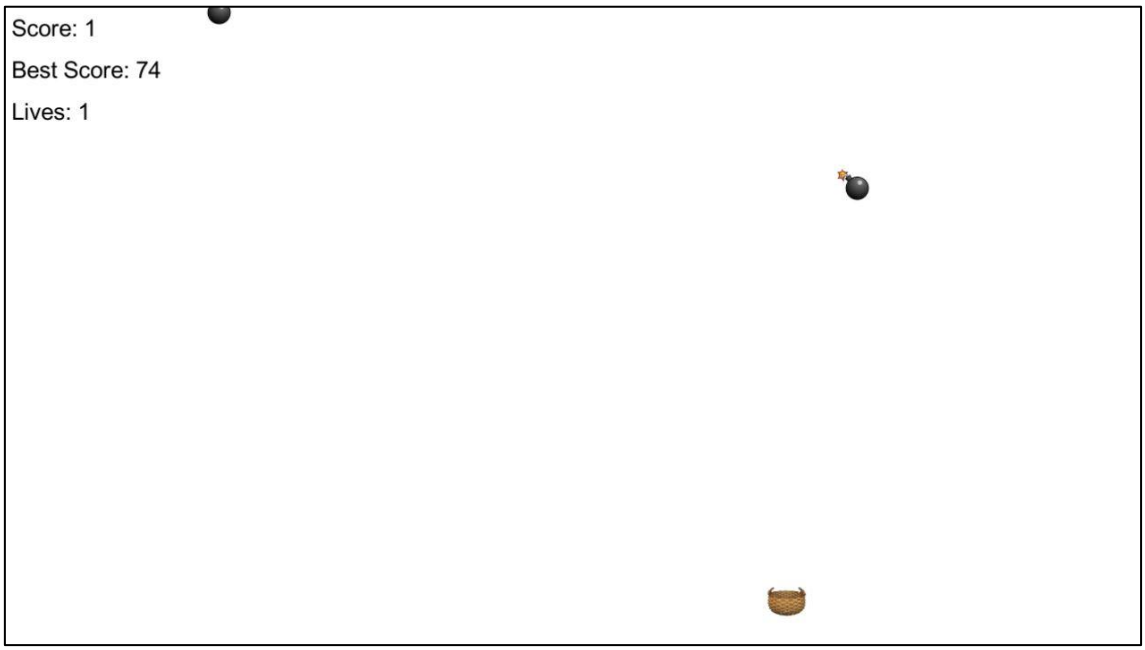
```
public void OnPlayButtonClick() {
    if(PlayBtnClick != null) {
        PlayBtnClick();
    }
}
```

KUVA 8. OnPlayButtonClick-funktio

Ensiksi funktiossa tarkistetaan, kuunteleeko mikään PlayBtnClick-tapahtumaa. Jos tapahtumalla on kuuntelijoita, silloin välittäjä nostaa PlayBtnClick-tapahtuman, jolloin kaikille tapahtuman kuuntelijoille tulee tieto siitä ja ne voivat reagoida siihen.

3.2.2 Pelinäkymä

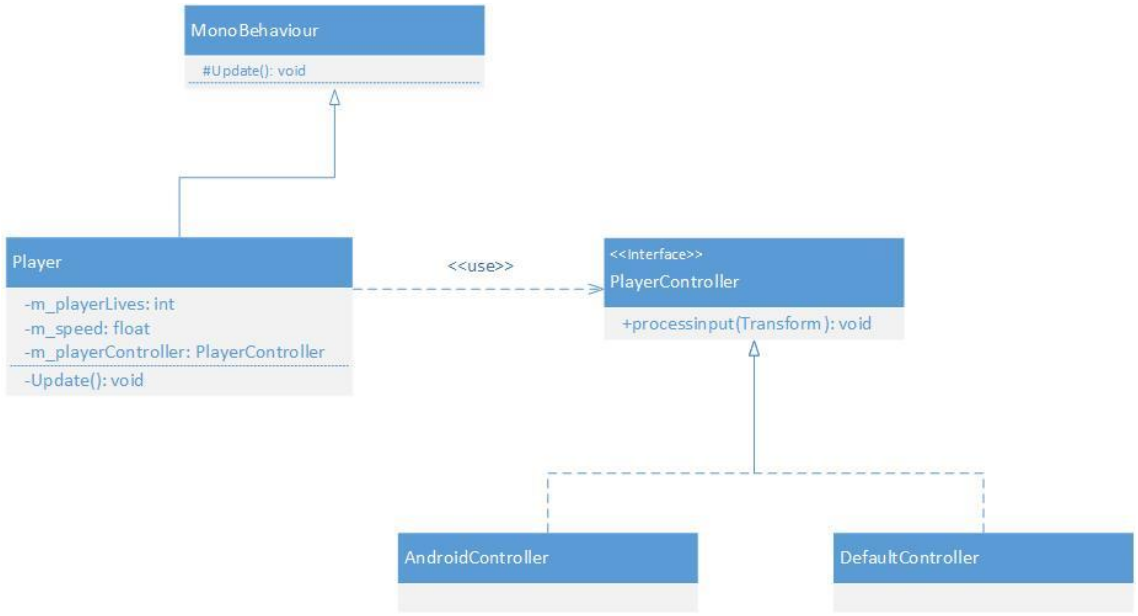
Pelinäkymä koostuu pelaajan ohjaamasta hahmosta, tippuvista esineistä sekä käyttöliittymästä, jossa näkyvät pelaajan saamat pisteet ja jäljellä olevat elämät sekä pelissä tehty paras tulos. Kuvassa 9 on esitetty kuva pelinäkymästä.



KUVA 9. Pelinäkymä

Pelaaja kerää ylhäältä putoavia pommeja koriin. Kiinniotetusta pommista pelaaja saa yhden pisteen ja ohimenneestä pommista hän menettää yhden elämän. Kun pelaajan elämät ovat loppuneet, hän häviää pelin ja voi halutessaan lisätä saamansa pisteet pistetauluun, jos pelaajan saama pistemäärä on riittävä.

Koriin on liitetty Player-skripti, joka kuvastaa pelaajaa pelin aikana. Kuvassa 10 näkyy Player-skriptin luokkakaavio.



KUVA 10. Player-skriptin luokkakaavio

Luokkakaaviosta nähdään, että Player-skriptillä on viittaus PlayerController-käyttöliittymään (interface). PlayerController-käyttöliittymän kautta päästään käsiksi käyttöliittymän toteuttavaan luokkaan, joka ohjaa koria.

Luokkakaaviosta nähdään, että PlayerController-käyttöliittymällä on kaksi eri toteutusta: Android-laitteella käytettävä AndroidController ja tietokoneella käytettävä DefaultController. Android-laitteella käytettävä toteutus käyttää kallistussensoria ja tietokoneella toimiva toteutus käyttää näppäimistön nuolinäppäimiä. Valinta siitä, kumpaa toteutusta käytetään, tehdään skriptin käännön aikana. Tämä on toteutettu käyttäen alustasta riippuvaa kääntämistä (Platform dependent compilation). Seuraavassa kuvassa on esitetty Player-skriptistä pätkä, jossa PlayerController-muuttujaan sijoitetaan joko AndroidController- tai DefaultController-olio riippuen alustasta.

Kuvan 11 koodissa alustasta riippuva koodi on kirjoitettu #if- ja #endif-tunnisteiden väliin.

```
#if UNITY_ANDROID
        m_playerController = new AndroidController(m_speed);
#endif

#if UNITY_EDITOR || UNITY_STANDALONE
        m_playerController = new DefaultController(m_speed);
#endif
```

KUVA 11. Alustasta riippuva ohjauksen valinta

Jos #if-tunnisteen jälkeen tuleva muuttuja on määritelty, tunnisteiden välissä oleva koodi käännetään (9).

Kuvassa 12 näkyy Player-skriptin Update-funktio.

```
void Update () {
    m_playerController.ProcessInput(transform);
    checkBounds();
}
```

KUVA 12. Player-skriptin Update-funktio

Update-funktiossa PlayerController-käyttöliittymän kautta kutsutaan joko kallistussensoria tai nuolinäppäimiä käyttävää ProcessInput-toteutusta alustasta riippuen. ProcessInput-funktio saa parametrina liikutettavan Transform-komponentin.

Ylhäältä putoavat pommit on toteutettu lisäämällä näkymään BombSpawner-niminen peliobjekti, jota ohjaa BombSpawn-niminen skripti. Skripti luo tietyn ajan välein uuden pommin, johon on liitetty fysiikkakomponentti sekä itsetehty DestroyBomb-skripti. Koska pommiobjektiin on liitetty fysiikkakomponentti, fysiikkamoottori vaikuttaa pommiin ja saa sen putoamaan.

Jos pommi on pudonnut pois pelialueelta, DestroyBomb-skripti kutsuu Mediator-olion OnBombMissed-funktiota ja tuhoaa pommin. Mediatorin OnBombMissed-

funktio nostaa BombMissed-tapahtuman. Player-skripti kuuntelee BombMissed-tapahtumaa ja reagoi siihen vähentämällä pelaajan elämiä sekä ilmoittamalla siitä Mediator-oliolle, joka nostaa sitä vastaavan tapahtuman. Player-skripti myös tarkistaa pelaajan jäljellä olevat elämät ja ilmoittaa Mediator-oliolle, jos elämät ovat loppuneet.

Pelinäkymässä on UiManager-objektiin liitetty pelinäkymän käyttöliittymää ohjaava GameUi-skripti. GameUi-skripti kuuntelee pelaajan elämien loppumista ilmaisevaa PlayerDead-tapahtumaa. Kuvassa 13 on esitetty PlayerDead-tapahtuman käsittelijäfunktio.

```
private void Instance_PlayerDead() {
    m_nameInputField.gameObject.SetActive(true);
    m_sendScoreBtn.gameObject.SetActive(true);

    Mediator.Instance.LifeLost -= Instance_LifeLost;
    Mediator.Instance.ScoreChanged -= UiManager_ScoreChanged;
}
```

KUVA 13. GameUi-skriptin PlayerDead-tapahtuman käsittelijäfunktio

Käsittelijäfunktion alussa laitetaan näkyviin pelaajan nimensyöttökenttä ja painike, josta pisteet voidaan lähettää palvelimelle, jos nimensyöttökenttään on syötetty nimi. Funktion lopussa lopetetaan LifeLost- ja ScoreChanged-tapahtuman kuuntelu.

Kuvassa 14 näkyy pelinäkymä, kun pelaaja on hävinnyt pelin.

KUVA 14. Nimensyöttökenttä

Kun pelaaja syöttää nimen nimensyöttökenttään, kentän alapuolella olevaan kenttään vaihdetaan tekstiksi ”Send”. Kuvassa 15 näkyy painikkeen klikkaustapahtuman käsittelijäfunktio.

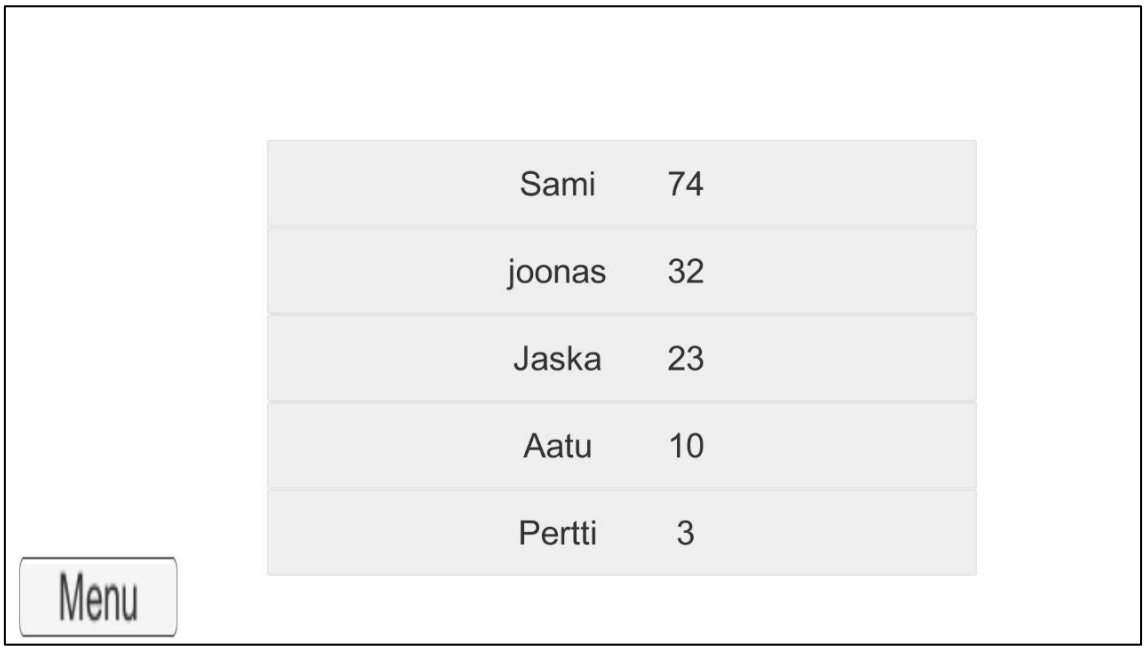
```
public void SendScoresbtn_Click() {
    string name = m_nameInputField.text;
    Debug.Log(name);
    if(name != "") {
        Mediator.Instance.OnSaveScore(name);
    }
    else {
        Mediator.Instance.OnBackToMenu();
    }
}
```

KUVA 15. Send-painikkeen klikkaustapahtuman käsittelijäfunktio

Kun painiketta painetaan, nimensyöttökentän teksti tarkistetaan. Jos kentän teksti ei ole tyhjä, pisteet tallennetaan ja palataan valikkoon, muussa tapauksessa palataan valikkoon.

3.2.3 Pistetaulukkonäkymä

Näkymä koostuu painikkeesta, josta pelaaja vai palata valikkoon, sekä viisi parasta tulosta näyttävästä pistetaulukosta. Kuvassa 16 näkyy pistetaulukkonäkymä.



KUVA 16. Pistetaulukkonäkymä

Kun näkymä on ladattu, pistetaulukkoa ohjaava skripti täyttää pistetaulukon. Tuloksia ei ole saatavilla voi tarkoittaa, ettei tuloksia ole saatu vielä ladattua. Tässä tapauksessa skripti käynnistää vuorottelualiohjelman (coroutine). Vuorottelualiohjelman määrittely näkyy kuvassa 17.

```
private IEnumerator isScoresReady()
{
    while (!m_scoresCreated)
    {
        if (GameManager.GetInstance().GetScores().Count > 0)
        {
            createScores();
        }
        else
        {
            yield return m_waitForSeconds;
        }
    }
}
```

KUVA 17. Vuorottelualiohjelma

Vuorottelualiohjelma tarkistaa puolen sekunnin välein, onko tuloksia saatavilla. Kun tulokset on saatu ladattua, pistetaulukko täytetään ja vuorottelualiohjelman suorittaminen lopetetaan.

4 YHTEENVETO

Opinnäytetyöni kolmannen osan aiheena oli toteuttaa mobiilipeli sekä siihen pilvipohjainen pisteiden tallennus ja lataaminen. Peli toteutettiin Unity-pelimoottorilla ja C#-ohjelmointikielellä. Pilvitietokantana toimii mLabin MongoDB, joka käyttää Mcrosoftin Azure-pilvipalvelua. Palvelin, joka tekee kyselyitä tietokantaan, toteutettiin NodeJS:llä ja se toimii Heroku-pilvialustalla.

Työn toteutus onnistui hyvin. Peli toimii hyvin mobiililaitteella, jolla sitä testattiin. Peliä oli myös mukava tehdä ja suunnitella. Skriptien väliset viittaukset pyrittiin pitämään mahdollisimman vähäisinä. Tämä onnistui käyttämällä C#:n tapahtumia sekä toteuttamalla erillinen skripti, joka vastaa kaikista tapahtuman nostoista. Koodista pyrittiin tekemään modulaarista käyttöliittymien avulla.

Pilvitietokannan toteutus onnistui hyvin ja sitä oli mukava tehdä. Pilvitietokannan toteutuksessa varsinainen koodaus oli NodeJS-palvelimen toteutus, muuten se oli vain oikeiden asetusten valitsemista. Palvelinta tehdessä oppi lisää web-pyynnöistä.

Peliä on tarkoitus jatko kehittää harrastusprojektina ja se on tarkoitus saada julkaistua Google Play -kaupassa. Pelissä ilmenee ajoittaista nykimistä, joka häiritsee pelattavuutta. Lisäksi peliin voisi lisätä grafiikkaa, jotta peli olisi miellyttävämpi silmälle. Peliin on myös hyvä saada vaihtelevuutta, jottei se käy tylsäksi pelata. Yksi vaihtoehto olisi laittaa eri nopeuksisia pommeja, jotka ilmestyvät sattumanvaraisesti. Peliin voisi myös tehdä eri tasoja, joilla esiintyisi erilaisia putoavia esineitä. Pelin kehittämisessä vain mielikuvitus ja omat taidot ovat rajana.

Lähteet

1. Build once, deploy anywhere. 2017. Unity Technologies. Saatavissa: https://unity3d.com/unity/multiplatform. Hakupäivä: 28.2.2017

2. Fear, Ed 2009. United they stand: Men united. Develop. Saatavissa: http://www.develop-online.net/analysis/united-they-stand/0116643. Hakupäivä 28.2.2017

3. Editor Version Release Dates. 2017. Unity Technologies. Saatavissa: http://web.archive.org/web/20141015144227/http://docs.unity3d.com/Manual/ReleaseDates.html. Hakupäivä: 1.3.2017

4. Unity 1.1: New Features. 2017. Unity Technologies. Saatavissa: https://unity3d.com/unity/whats-new/archive. Hakupäivä: 1.3.2017

5. Fear, Ed 2009. United they stand: The Macs factor. Develop. Saatavissa: http://www.develop-online.net/analysis/united-they-stand/0116643. Hakupäivä: 2.3.2017

6. Unity Roadmap. 2017. Unity Technologies. Saatavissa: https://unity3d.com/unity/roadmap. Hakupäivä: 2.3.2017

7. Bibby, Brett 2017. Unity 5.6 wraps Unity 5 cycle, what's next in 2017. Unity Technologies. Saatavissa: https://blogs.unity3d.com/2016/12/13/unity-5-6-wraps-unity-5-cycle-whats-next-in-2017/. Hakupäivä: 2.3.2017

8. Bugnion, Pascal – Manivannan, Arun – Nicolas, Patrick R. 2017. Scala: Guide for Data Science Professionals. Birmingham: Packt Publishing Ltd.

9. Platform dependent compilation. 2017. Unity Technologies. Saatavissa: https://docs.unity3d.com/Manual/PlatformDependentCompilation.html. Hakupäivä: 17.3.2017